NASA-CR-193239

Old Dominion University Research Foundation

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

171915
1 62

BUILDING A GENERALIZED DISTRIBUTED SYSTEM MODEL

By

R. Mukkamala, Principal Investigator

Annual Report

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23681

Under
Research Grant NAG-1-1114
Wayne Bryant, Technical Monitor
ISD-Systems Architecture Branch

June 1993

N93-29219
Unclas
G3/61 0171915

(NASA-CR-193239) BUILDING A GENERALIZED DISTRIBUTED SYSTEM MODEL (Old Dominion Univ.) 62 p

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

BUILDING A GENERALIZED DISTRIBUTED SYSTEM MODEL

By

R. Mukkamala, Principal Investigator

Annual Report

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23681

Under
Research Grant NAG-1-1114
Wayne Bryant, Technical Monitor
ISD-Systems Architecture Branch

Submitted by the
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, Virginia 23508-0369

June 1993

Building a Generalized Distributed System Model

R. Mukkamala
E.C. Foudriat

Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529-0162.

Annual Report and Renewal Request

Abstract

The key elements in the 1992-93 period of the project are:

- Extensive use of the simulator to implement and test
  - Concurrency control algorithms
  - Interactive user interface, and
  - Replica control algorithms
- Investigations into the applicability of data and process replication in real-time systems, and

In the 1993-94 period of the project, we intend to:

- Concentrate on efforts to investigate the effects of data and process replication on hard and soft real-time systems. Especially we will concentrate on the impact of semantic-based consistency control schemes on a distributed real-time system in terms of
  - Improved reliability
  - Improved availability
  - Better resource utilization, and
  - Reduced missed task deadlines
- Use the prototype to verify the theoretically predicted performance of locking protocols, etc.

1 Introduction

In the 1992-93 proposal, we proposed to test the simulator more extensively. This has been achieved by using it to test the performance of replica control algorithms and concurrency control algorithms. In addition, we have now constructed a user interface using which it is possible for a user to specify the selection of nodes (by their machine name) and the way they need to be connected. In addition, the user may specify the location of the code for execution at these nodes. The simulator would then start the required execution.

The investigations into the use of replication in distributed real-time systems has been preliminary and no reports or papers have yet been published. However, there is strong evidence that this area will be a strong candidate for future system architectures. For this reason, we propose to further pursue this approach in the coming year. This should lead to several conference publication and possibly journal submissions.

In this report, we summarize our progress in these areas and then describe the proposed work for 1993-94.

2 Distributed System Prototype

As stated above, we have used the current prototype tool to implement and test concurrency control algorithms and replica control algorithms. Here, we will summarize these efforts.

2.1 Concurrency Control Algorithms

Using the previously developed modules of the prototype, we have successfully implemented and tested an optimistic concurrency control algorithm using time stamping. The algorithm is optimistic in the sense that it relies mainly on transaction backup as a control mechanism hoping that conflicts between transactions are minimal. In this case, a transaction consists of three phases: read-phase, validation-phase, and a write-phase. The details of the implementation, and the way the prototype modules are combined are discussed in an attached report.

2.2 Replica Control Algorithms

Replica control is necessary to assure mutual consistency in distributed systems, when a high degree of fault-tolerance needs to be provided. As part of this effort, a replication algorithm is implemented using a weighted voting method, where a quorum of votes must be obtained to execute read or write operations. The Global transaction module (GTM) of the prototype has been slightly modified to meet the requirements of the replica control algorithm. The details of the work are described in an attached report.

3 Interactive User Interface to the Prototype

The main objective of this project were to implement a system which will automatically establish a customized configuration of workstations connected on a network (distributed system). Some of the parameters which the user may wish to input to the system are: names of sites to be included in the system; the algorithm/program code to be run on the system, the logical topology by which the chosen sites are to be connected; and the logfile where the output file should be directed to. The system has also been used to test a token-ring protocol specified via the interface. The details of the effort are described in an attached report.

4 Distributed Real-time Systems: Current efforts

Due to the importance of reliability and timeliness in real-time systems, the application of distributed systems in this area is now well recognized. In this context, we started looking at both hard and soft real-time systems (centralized) and the work in distributed database systems (non-real time). We find that by effectively combining ideas from these two systems, we can build distributed real-time systems, especially using the data and process replication ideas.

5 Proposed Research Efforts in 1993-94

During the next grant period (August 1993 - July 1994), we propose to mainly concentrate on the issues related to distributed real-time systems. Especially, we propose to study and solve the following problems.

- How can data replication be effective used to improve reliability, availability, and reduce number of tasks missing deadlines?

  This effort will require the study of currently existing non-replicated algorithms, and solving some of the system bottlenecks by employing replication. While the answer for improvements seems to be obvious, how to maximize the benefits is not at all obvious.

- Can Replication be also used to reduce the non-deterministic delays involved in the communication subsystem? If so what is the added cost, and other effects due to increased load?

  Several efforts in minimizing the non-determinism in the communication network delay, especially for high priority traffic are already in progress and published in literature. However, none have studied replication as a means to achieve it. We propose to consider replication of channels, replication of transmissions, and replication of servers to achieve it. Obviously, we need to take into consideration the cost involved with the proposed schemes.

- What is the effect of replica consistency requirements on the performance of the system? Obviously, the more stringent the requirements, the worse will be the performance. But we would like to study this aspect in a more comprehensive manner.

- Develop newer scheduling algorithms that can incorporate the new consistency requirements.

- Further develop and use the current distributed system prototype to test and implement distributed algorithms. It will also be used in the distributed systems class to be offered in Fall 1993.

Implementation of an Optimistic Concurrency Control Algorithm using Timestamps

Sastry.A.V.R.R
Shubhangi Kelkar
Pradeep Sankaranthi

December 11, 1992

Introduction :

Concurrency control is the activity of coordinating concurrent accesses to a database in a multiuser database management system(DBMS). Concurrency control permits users to access a database in a multiprogrammed fashion while preserving the illusion that each user is executing alone in a dedicated system. The concurrency control problem is complicated in distributed DBMS(DDBMS) because users may access data stored in different computer in a distributed system and a concurrency control mechanism at one computer cannot instantaneously know about interactions at other computers. Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this project an optimistic concurrency control using timestamps is implemented. It is optimistic in the sense that they rely mainly on transaction backup as a control mechanism hoping that conflicts between transactions will not occur. This approach has the advantage that it is completely general, applying equally well to any shared directed graph structure and associated access algorithms. Since locks are not used, it is deadlock free. The idea behind this optimistic approach is quite simple and may be summarized as follows.
Any transaction consists of two or three phases:a read phase, a validation phase, and a possible write phase. During the read phase, all writes take place on local copies of the nodes to be modified. Then, if it can be established during the validation phase that the changes that the transaction made will not cause a loss of integrity, the local copies made are global in the write phase. In the case of a query, it must be determined that the result the query would return will actually be correct. The step in which it is determined that the transaction will not cause a loss of integrity (or that it will return the correct result) is called validation.

If validation fails the transaction will be backed up and start over as again as a new transaction. Thus a transaction will have a write phase only if its previous validation succeeds.

Timestamp ordering (T/O)is a technique whereby serialization order is selected apriori and transaction execution is forced to obey this order. Each transactionis assigned a unique Timestamp by its Transaction Manager. The TM attaches the timestamp to all reads and writes issued on behalf of the transaction, and Data Managers are required to process conflicting operations in timestamp order.

Implementation:

Message formats and the sequence of events/messages are shown in the table 1. Fig 1 shows the interaction between various modules.

The way this system functions is as follows. User submits his transaction which involves a series of Reads and/or Writes to the User Transaction Manager(UT). UT verifies the syntax of the transaction and passes this request to Global Transaction Manager (GTM) with a timestamp attached to it. GTM takes this transaction and assigns a unique transaction ID to this transaction. It then enqueues the transaction in its transaction queue in the increasing timestamp order. GTM maintains the status of both local and remote transactions at any time. GTM then parses the transaction and divides the transaction into subtransactions. Each subtransaction status is maintained in a subtransaction queue under the corresponding transaction queue. The execution of transaction- now involves execution of these subtransactions. GTM sends a request to Replica Control (RC) for location and quorum(R/W) information about each object. All the messages that is sent to RC bear a timestamp on them. RC maintains a list of all sites participating in the functioning of the overall system, information about the objects residing at the site viz. object_id, site_id, votes required for read or write.

RC then sends a reply to GTM stating the site(s) at which the data object in question will be available. GTM takes this information and then send a message to local LTM for execution of subtransactions which are local. If the subtransaction requires an object that is remote to this site then GTM sends a message to remote GTM for execution of the subtransaction at that site. At the level of LTM it is not possible to distinguish between local and remote transactions. If the subtransaction is a READ operation then LTM sends a request for Physical read to Resource manager(RM) where the actual database is located. RM acts as a scheduler for reads and writes for read and write requests. RM replies to LTM with a Physical Read done message. If however, the operation involves writing the data object, then LTM sends a message to Local Transaction Recovery Manager (LTRM) asking it to write the data item *logically*. This is different from physical write which is not done until a *commit* message is received. LTRM on receiving the logical write request store the object_id and corresponding value in a structure. It then sends a message to RM asking for the timestamp of the item to be written. RM reads the timestamp of the item and sends it to LTRM. LTRM preserves this timestamp value in corresponding subtransaction's data structure. Then LTRM sends a message Logical Write Done to LTM. LTM propagates this

message to GTM. On receiving/not receiving successful write done(logical) from all the sites (quorum limit), the GTM make a decision to commit/abort the subtransaction. A two-phase commit protocol is used to ensure that either all sites "commit" or "abort" a transaction, thus maintaining data consistency. If GTM receive all the logical write done messages for all its write subtransactions it sends a commit message to LTM. LTM passes this message to LTRM. LTRM now checks if the timestamp associated with the item is same as the timestamp of the item at the time of logical write. For this to accomplish, it sends a message to RM asking for timestamp again. RM replies with item's timestamp. LTRM checks the received timestamp with the timestamp of the item stored in its data structure. If both are same it issues a message Physical Write request to RM. This procedure ensures consistency of the data item. RM replies with a Physical write done message after it modifies the value of the data object in the actual database. Having received the Physical Write Done message from RM, LTRM sends a commit done message to LTM which passes the same to GTM. GTM then lets the UT know that the submitted transaction is successfully done.

The original code which implemented two phase locking as a means of obtaining serializability is modified to suit our requirements in the following manner.

UT

- Timestamps are included in all the messages that are sent
- Osn is updated eachtime a message is sent/received
  (these steps are repeated in all the modules)

GTM:

- Timestamp is taken from message received from the UT/Remote GTM and put the message in the queue (addtransaction()) based on timestamp ordering.
- After it get a reply from the RC, GTM sends a message to LTM instead of GCCM (as in the original code).

GCCM and LCCM

- completely eliminated since we are not using two phase locking.

RC

- All messages are embedded with a logical clock field.

LTM

- In the LTM.h timestamp field is added to the _ltmstruct structure for each item.

LTRM

- In the LTRM.h we added numAcksExpected and aborted fields to check for the consistency of the original data item.
- All calls to LCCM are eliminated because none of the locks need be acquired before.

- Four more cases are added.
  READ_ITEM_TS_FIRST, READ_ITEM_TS_AGAIN, ITEM_TS_FIRST_TIME, ITEM_TS_SECOND_TIME

  These cases are introduced to check if the item in question is modified by any other transaction before committing this transaction.

RM

- Each data object will have a timestamp associated with it. Everytime, this object is modified the timestamp field is updated to the logical clock value at that time.
- Two more cases are added to handle read requests from LTRM.
  READ_ITEM_TS_FIRST and READ_ITEM_TS_AGAIN

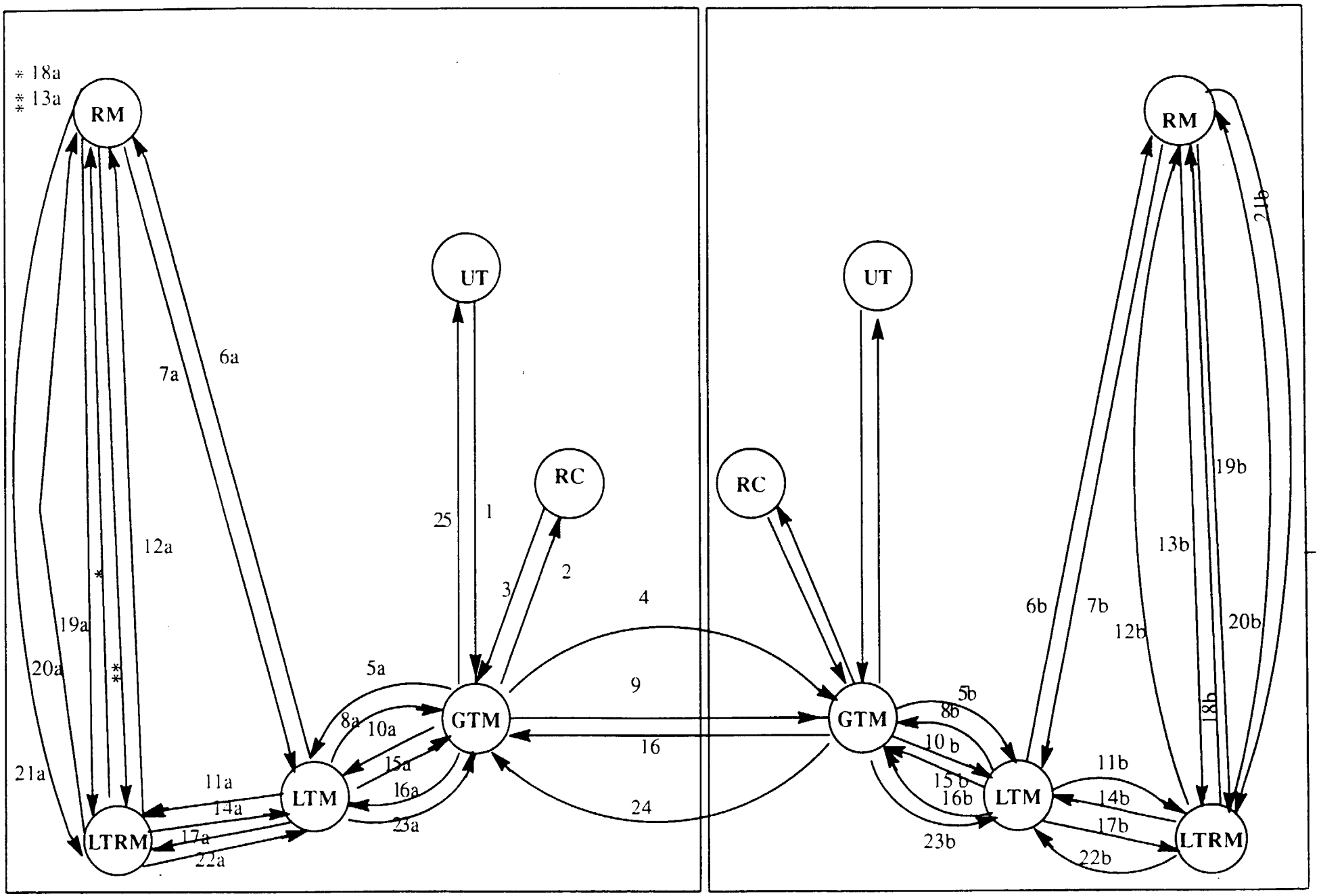
Opcodes.h

- Four new opcodes described above are included in the opcodes table.

Conclusions:

For rigorous testing of the current code a small delay can be introduced in the transactions submitted so that some confilicts can be simulated. Alternatively, original code can be refined so that each of the transaction managers act as servers instead of waiting in a tight loop for the messages to come. Thus the server will fork off a process for each of the messages it received.

Data Control Flow Diagram Of The Distributed System Prototype



Distributed Systems Project Operations and Message formats

December 9, 1992

Step#	Operation
1	UT module sends a transaction to GTM. The transaction may look like: Read A Read B Write C Write D where each Read/Write is called a "Subtransaction". The message format is : [TIME_STAMP][USER_TRANSACTION_REQUEST(L)][USER_TRANSACTION_BEGIN(I)] [USER_ID(I)][READ_OP(I)][ITEM_ID(I)][WRITE_OP(I)] [ITEM_ID(I)][DATA(I)]...[USER_TRANSACTION_END(I)]
2	GTM enqueues the incoming transactions on the basis of the time stamp and then makes subtransactions for every Read/Write of an item (eg. A, B) and sends a request to RC for complete knowledge of replication and qourum needed for R/W. The message format is : [TIME_STAMP][QOURUM_READ(WRITE)_REQUEST(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][R/W_ID(I)]
3	RC finds an optimal list of sites needed for R/W qourum of an item in a subtransaction. RC sends this list to GTM. The message format is :
a.	[TIME_STAMP][QOURUM_READ(WRITE)_REPLY(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][R/W_ID(I)][QUORUM(I)][NUM_SITES(I)] [SITENAME1(I,S)][VOTE1(I)]...
b.	[QOURUM_READ(WRITE)_REFUSED(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][R/W_ID(I)]
5a	
a.	If a subTx is a READ(local): A "read" operation is sent to LTM. The message format is : [TIME_STAMP][READ_REQUEST_LOCAL(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]
b.	If a subTx is a WRITE(local): A "write" operation is sent to LTM. The message format is : [TIME_STAMP][WRITE_REQUEST_LOCAL(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][VERSION(I)][DATA(I)]

Step#	Operation
4	If a subTx is a READ(remote): A "read" operation is sent to remote GTM. The message format is : [TIME_STAMP][READ_REQUEST_REMOTE(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)] If a subTx is a WRITE(remote): A "write" operation is sent to remote GTM. The message format is : [TIME_STAMP][WRITE_REQUEST_REMOTE(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][VERSION(I)][DATA(I)]
5b	same as 10a (Now request is local).
6a,6b	Local READ operation: LTM passes the read operation to RM of the site. The message format is : [TIME_STAMP][PHYSICAL_READ_REQUEST(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]
7a,7b	Local READ reply from RM to LTM. The message format is : [TIME_STAMP][PHYSICAL_READ_DONE(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][DATA(I)]
8a,8b	Local READ done reply from LTM to GTM: The message format is : [TIME_STAMP][READ_DONE_LOCAL(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][DATA(I)]

Step#	Operation
9	Remote READ done is passed back to the GTM at which it originated. The message format is : [TIME_STAMP][READ_DONE_REMOTE(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][DATA(I)]
10a,10b	Local WRITE operation: GTM passes the write operation for an item mentioned in a subTx to its LTM. The message format is : [TIME_STAMP][WRITE_REQUEST_LOCAL(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][DATA(I)]
11a,11b	LTM passes WRITE operation to LTRM : The message format is : [TIME_STAMP][LOGICAL_WRITE_REQUEST(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][DATA(I)]
12a,12b	Operation
	LTRM sends a request to RM for the item's original timeStamp. [TIME_STAMP][READ_ITEM_TS_FIRST][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]
13a,13b	Operation
	RM sends the reply by stuffing the item's timeStamp in the message. [TIME_STAMP][ITEM_TS_FIRST_TIME][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)][ORIGINAL_TIMESTAMP]
14a,14b	LTRM sends "write done" reply to LTM. LTRM actually stores the value in a datastructure along with the item's original timeStamp it received in the earlier message (Physical write is still not done.) The message format is : [TIME_STAMP][LOGICAL_WRITE_DONE(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]

Step#	Operation
15a,15b	A "prepared" message is sent to GTM by LTM when a "write done reply" is received. The message format is : [TIME_STAMP][PREPARED(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]
16a,16b	When correct # of "prepared" messages are/(are not) received by GTM for all write subTx's of a Tx, a commit/abort message is sent to local LTM for updates(final physical WR) and all GTMs(remote) which are involved in updates. local message formats are(commit and abort)(GTM to LTM): [TIME_STAMP][TRANSACTION_COMMIT_LOCAL(L)][USER_TRANSACTION_ID(I)] [TRANSACTION_ABORT_LOCAL(L)][USER_TRANSACTION_ID(I)]
16	Remote message formats are(commit and abort)(GTM to GTM): [TIME_STAMP][TRANSACTION_COMMIT_REMOTE(L)][USER_TRANSACTION_ID(I)] [TRANSACTION_ABORT_REMOTE(L)][USER_TRANSACTION_ID(I)]
17a,17b	Commit or Abort is passed to LTRM by LTM The message format is : [TIME_STAMP][COMMIT_LOCAL(L)][USER_TRANSACTION_ID(I)] [ABORT_LOCAL(L)][USER_TRANSACTION_ID(I)]
18a,18b	LTRM asks RM for the item's timeStamp again. The message format is : [TIME_STAMP][READ_ITEM_TS_AGAIN][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]
19a,19b	RM sends a reply by including the item's timeStamp in the message The message format is : [TIME_STAMP][READ_ITEM_TS_AGAIN][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]
20a,20b	LTRM sends a message to A. RM if the timeStamp received matches with the earlier timestamp it got in reply to ITEM_READ_TS_FIRST message. The message fromat is: [TIME_STAMP][PHYSICAL_WRITE_REQUEST][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)] B. if the timestamps of item it received in response to READ_ITEM_TS_FIRST and READ_ITEM_TS_AGAIN don't match it sends an abort message to LTM. [TIME_STAMP][COMMIT_DONE][USER_TRANSACTION_ID(I)]

Step#	Operation
21a,21b	The "Physical write done" is sent back LTRM. The message format is : [TIME_STAMP][PHYSICAL_WRITE_DONE(L)][USER_TRANSACTION_ID(I)] [SUB_ID(I)][ITEM_ID(I)]
22a,22b	"Commit Done" is sent back to LTM. The message format is : [TIME_STAMP][COMMIT_DONE(L)][USER_TRANSACTION_ID(I)]
23a,23b	"Commit Ack" is sent back to GTM. The message format is : [TIME_STAMP][COMMIT_ACK(L)][USER_TRANSACTION_ID(I)]
24	Remote GTM sends the "Commit Ack" back to original GTM The message format is : [TIME_STAMP][COMMIT_ACK_REMOTE(L)][USER_TRANSACTION_ID(I)]
25	GTM when receives enough # of commits ACKs from all involved sites, it announces "User Transaction Done" to UT The message format is : [TIME_STAMP][USER_TRANSACTION_DONE(L)][USER_TRANSACTION_ID(I)]

Distributed Systems Replica Control Project

A Group Project for CS 763

Group Members: Rongli Jiang, Pat Mullally, Kent Stevens

Abstract

Replica Control is necessary in distributed systems to assure mutual consistency and provide a degree of fault tolerance.[1,2] *If data is not replicated among the nodes of the distributed system, a node failure can be responsible for significant system degradation. In this project, a replication algorithm is implemented using a weighted voting method, where a quorum of sites and votes must be obtained to execute a read or write transaction.*

Introduction

The purpose of this project is to design and implement a Replica Control (RC) algorithm which is based on a weighted voting scheme[3]. In order to obtain permission to commit a read or write transaction, a number of votes representing a quorum of votes must be established according to the following expression: *read quorum + write quorum > total number of votes*. A majority group is then determined by the expression[2] $\frac{v_{site\ 1} + v_{site\ 2} \ldots v_{site\ n}}{2} + 1$. The weighted voting scheme improves the overall probability that a read or write quorum can be achieved under conditions where replicated copies of data have been limited. For simplicity and to easily observe operation of the algorithm, voting weights are statically assigned to each site and stored in linked list data structures. Voting quorum values are maintained at each site by transaction number, and are loaded during node initiation. The necessary data structures used to compose and decompose messages are referenced when a transaction occurs in the system.

The algorithm has been designed to be extensible and compatible with the full implementation of the distributed network system. In the distributed network implementation, voting assignments can be deterministically assigned based on factors such as the states of a site or by parameters which may include reliability of the site or resources available to the site.

Assumptions

In order to successfully implement this project within the limited time available, various assumptions were made to limit the project scope. The assumptions are:

- The replica control algorithm will not consider partitioning.
- Voting assignments of the sites are established *a priori* to system operation and are statically maintained during system operation.

- Only one transaction will be processed by the distributed replica control algorithm at a time. Since our model does not contain any provisions for concurrency control, multiple transactions could possibly cause conflicts which would affect algorithm performance.
- Transaction messages are sent and received through statically assigned socket numbers. Dynamic port assignments are not considered in this implementation, however would be a feature which would be implemented in a distributed system.
- The program provides limited error recovery for timing out quorum request responses. A time-stamp file has been incorporated in the **TRANSACTION** data structure in which the initiation time of the transaction is recorded.
- This project uses a minimally functional Global Transaction Manager (GTM) to process, format, and communicate user inputs to the RC module. All data structures used in this project are pointer based and are easily extensible. The structures can be easily modified to handle greater numbers of transactions and more complex message structures.
- Some of the functions performed by the GTM have been transferred to the Replica Control (RC) module for this project. A Simple Global Transaction Manager (SGTM) has been implemented for this project.
- The user will input transaction ID from the keyboard, which will be processed through the SGTM to the RC module. An extensive user interface was not attempted, since the focus of this project was to implement the RC algorithm.

Message Formats

The following describe the message formats used for passing messages between the distributed sites in this system:

SGTM ➡ RC

- [READ_REQUEST][TRANSACTION_ID][ITEM_ID]
- [WRITE_REQUEST][TRANSACTION_ID][ITEM_ID]

SRC ➡ SGTM

- [TRANSACTION_COMPLETE][TRANSACTION_ID][ITEM_ID]
- [TRANSACTION_ABORT][TRANSACTION_ID][ITEM_ID]

RC ➡ RC (Remote)

- [LOCK_REQUEST][TRANSACTION_ID][ITEM_ID]
- [LOCK_REFUSED][TRANSACTION_ID][ITEM_ID]
- [LOCK_GRANTED][TRANSACTION_ID][ITEM_ID]

Data Structures

There are four basic data structures used in the project. The tables and figures included in this section show the data structures as they relate to the elements of a transaction. Figure 1 shows the **TRANSACTION** structure. Figure 2 shows the **logical item** structure which maintains replica information. Figure 2 also shows the **physical item** structure which contains the site number and the number of votes for each site. Figure 3 shows the **site_info** structure which contains the data about the locking and voting status of a site.

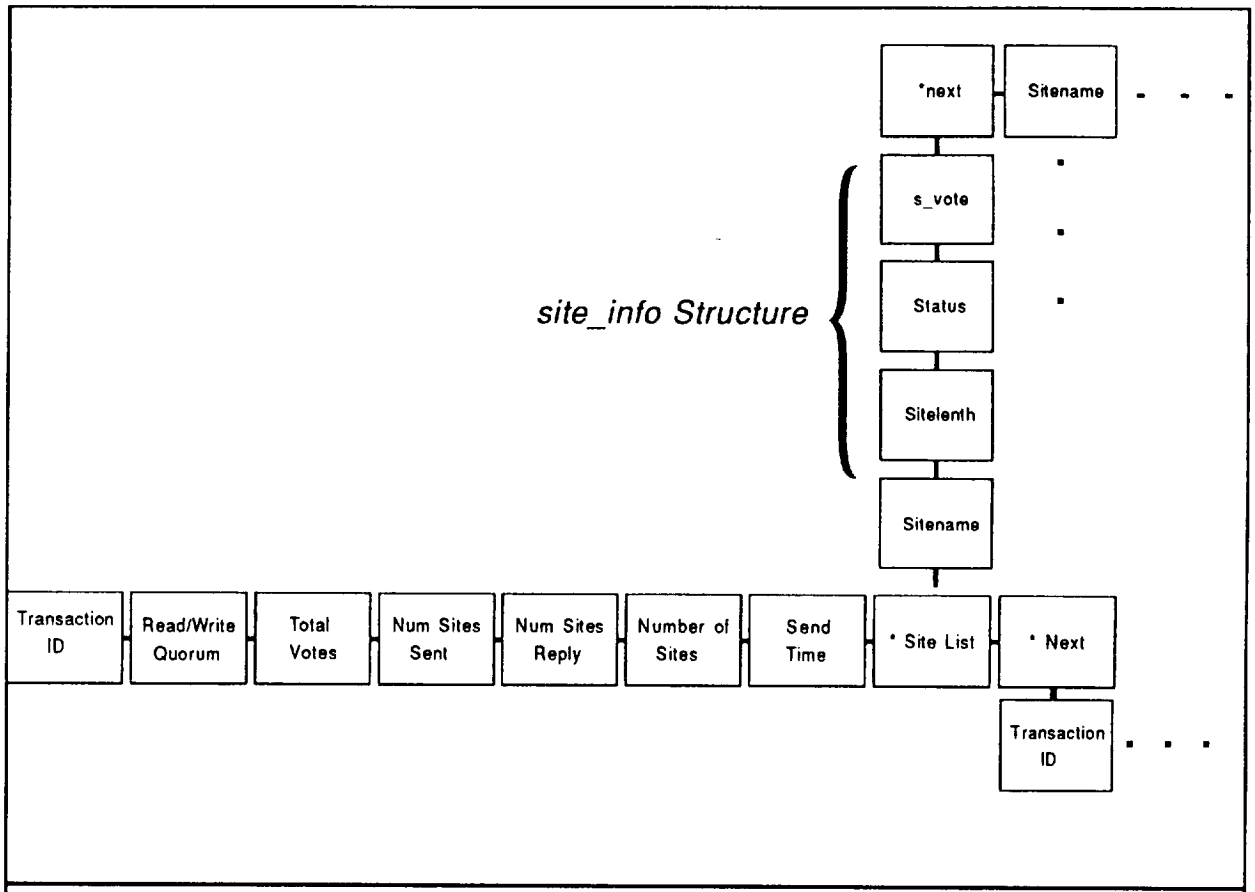


Figure 1. TRANSACTION Data Structure

TRANSACTION Data Structure	
Field	**Field Description**
trans_id	The transaction identification number. This number is an integer which is used to identify and synchronize transactions as they are received and processed by the sites.
rw_quorum	Read or Write quorum. This integer specifies whether the node is attempting to achieve a read or write quorum.
total_votes	Total votes is an integer obtained by the transaction.
num_sites_sent	This value is incremented when a message is sent to a remote site.
num_sites_reply	Calculated value obtained by adding the message obtained from each site to obtain the total number of sites which replied.
num_sites	The total number of sites which have a specified data item.
send_time	This is the system time when the transaction was initiated.
*site_list	This is a pointer to the site list which is used to determine the site_info data for a specific site and described in the site_info table.
*next	This is a pointer to the next transaction.

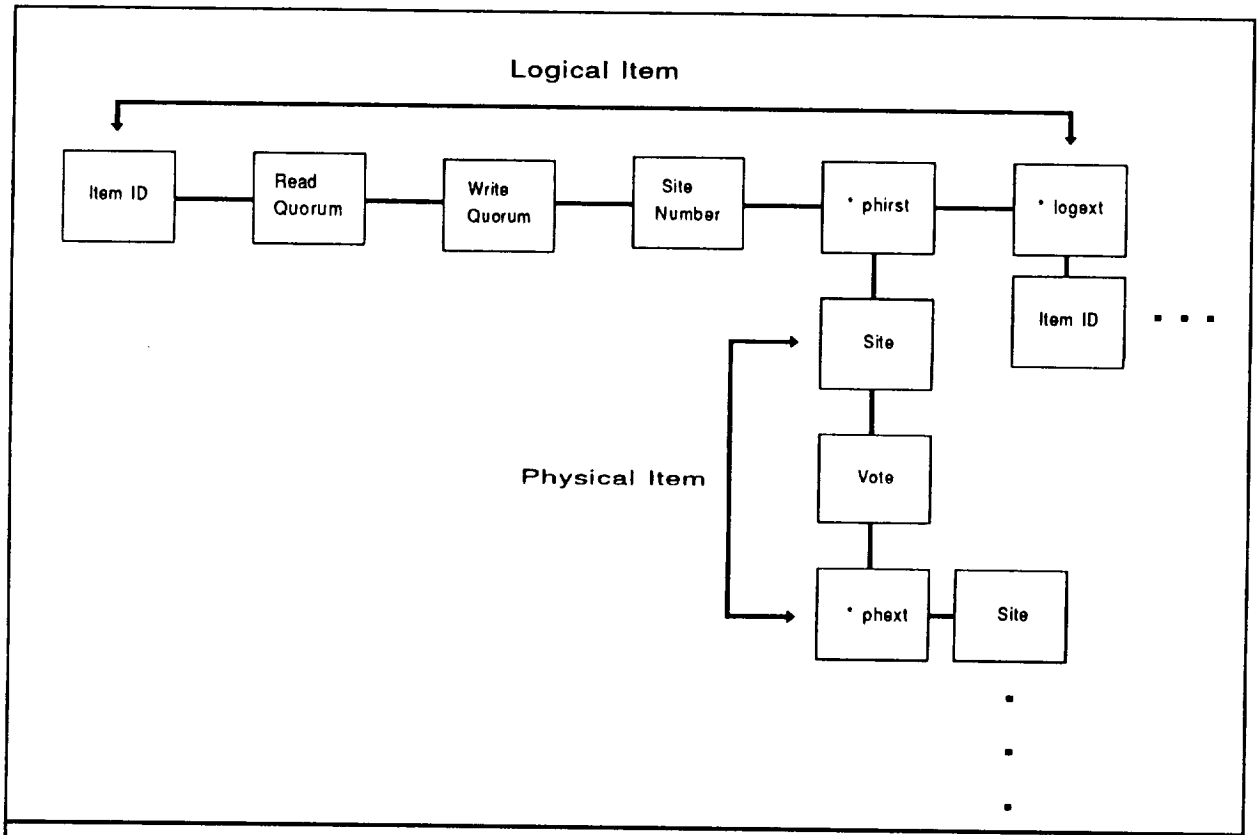


Figure 2. Relationship of Physical and Logical Items

physical_item Data Structure	
Field	**Field Description**
site	Referenced by the pointer *phirst of the logical_item data structure. Integer identifying the site number.
vote	Integer identifying the number of votes for the site.
*phext	Pointer to the next physical item.

logical_item Data Structure	
Field	**Field Description**
item_ID	Corresponds to the item_id number. Used as the primary key to reference the replica data.
r_quorum	Read quorum obtained by calculating the majority number of votes by applying the equation $\frac{Vsite\ 1 + Vsite\ 2 \ldots Vsite\ n}{2} + 1$.
w_quorum	Obtained by calculating the majority number of votes by applying the equation $\frac{Vsite\ 1 + Vsite\ 2 \ldots Vsite\ n}{2} + 1$.
site_num	Holds the number of copies for this item which are in the system.
*phirst	Pointer to the Physical Item data structure.
*logext	Pointer to the logical_item data structure.

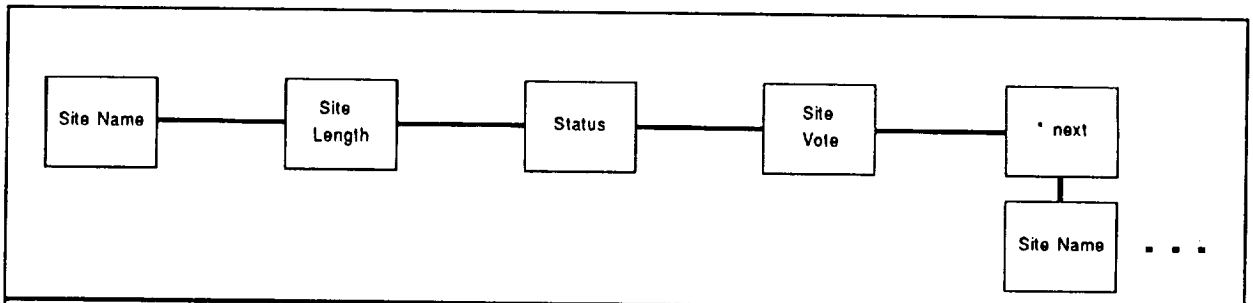


Figure 3. Site_info Data Structure

site_info Data Structure	
Field	**Field Description**
sitename	Pointed to by the site_list pointer in the TRANSACTION data structure. Contains the site name for the transaction.
sitelenth	An integer representing the string length of the site name.
status	Indicates whether a site has replied to a transaction request.
s_vote	Number of votes at the local site.

Implementation

General

Two major modules were needed to provide a functional demonstration of the RC algorithm. An SGTM was developed to provide a user interface and to communicate transaction requests to the RC module. In a fully implemented distributed system, the SGTM would also handle all of the communication between local and remote sites and would handle many different types of local messages between other functional modules of the system such as concurrency control, deadlock detection, etc. To confine the scope of this project, the RC module was designed to communicate with remote sites instead of the SGTM module as shown in Figure 4.

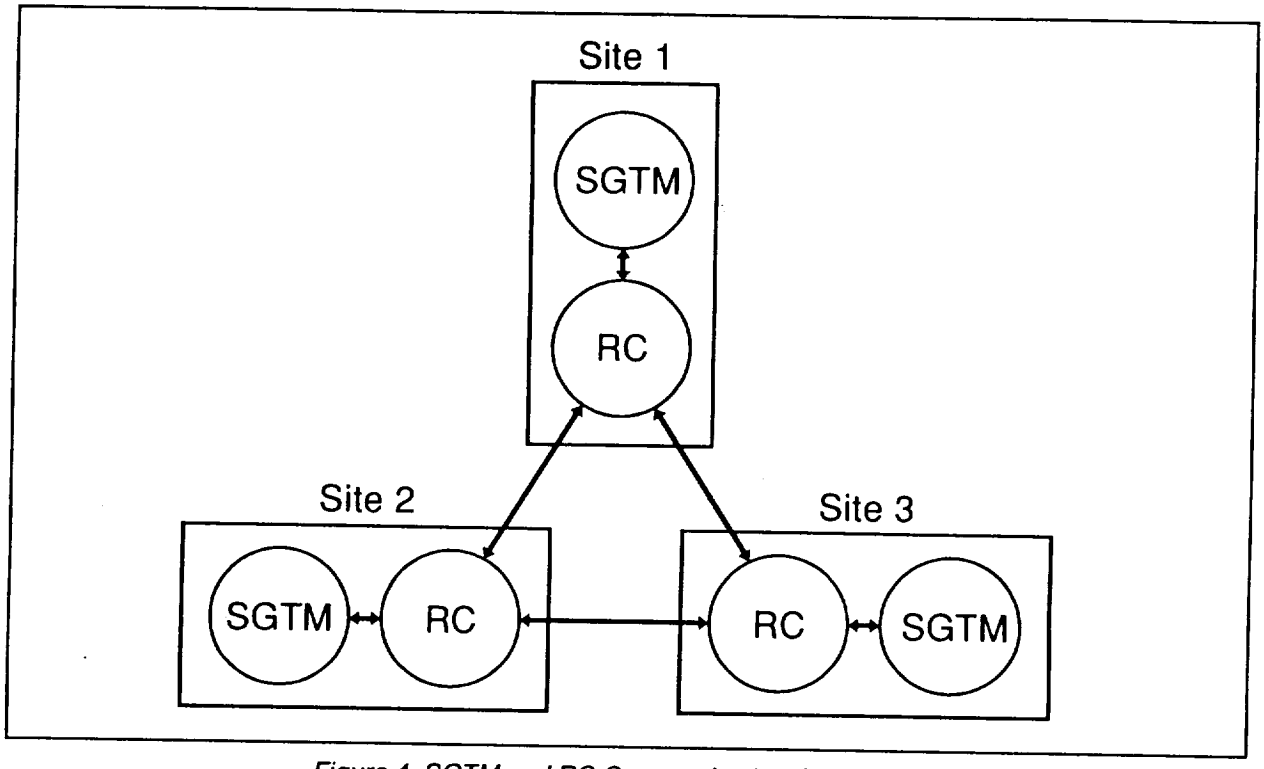


Figure 4. SGTM and RC Communication Configuration

The SI InterProcess Communication Library was used to transmit and receive messages between local and remote sites. Limited exception handling functions were added to provide some degree of robustness to the system.

Functional Description

The following is a description of the operation of the main functions and procedures of the **RC** program module.

main(argc,argv)

The **main** procedure obtains operator inputs for the module port and SGTM port and assigns them to the first two positions of the **argv** array. The procedure call **gethostname** is to the SI library which retrieves the name of the host to a string array

which is then mapped to an integer value. The **siConvertName** procedure call takes the module name and converts it to an integer, and returns an integer which is checked to determine if the operation was successful. The **init_rc_datatables()** procedure creates the replica table and a locking table by reading the **_replicadata** file and **_lockdata file** from the local site. The replica data table is initialized by reading from the file **itemid, site number**, and the **number of votes** for each site. The locking table is initialized by reading into the data structure the **itemid** and the **lock** status (**0** not locked, **1** locked).

The sockets for SGTM and RC are then initialized by the **init_rc_socket** procedure. This procedure initializes the RC and SGTM with the port numbers entered by the user at the keyboard.

Following the initialization functions, the main function enters an infinite while loop. The loop calls the SI library function **siReceivefrom()** function which continuously polls the ports received data. In this project, there are two possibilities for receiving data: data received from a remote RC at another site, or data received from the local SGTM. When a message is received, the message is first compared to the **moduleport** variable and if the comparison is true, the procedure **process_from_remote_RC()** is called. If the **moduleport** comparison is false, the message is compared to the **GTMport** variable. If this comparison is true, the function **process_from_local_GTM** procedure is called.

process_from_remote_RC(sd,msg,bytes)

This procedure processes message requests for read/write transactions, or replies to a local request to read/write a data item. The procedure first parses the message into the **op_code**, **trans_id**, and **item_id** fields of the transaction. A switch statement determines what to do based on the message op_codes **LOCK_REQUEST**, **LOCK_GRANTED**, and **LOCK_REFUSED**.

process_from_local_GTM(msg,bytes)

This procedure is used by the RC module to find the replica data for the transaction **itemid**. The procedure first parses the **op_code transid** and **itemid** from the message, and then checks to determine if the **itemid** exists. If it does not, a message is sent to the SGTM to abort the transaction. If the transaction is valid, it is appended to the head of the transaction queue. A **LOCK_REQUEST** is then sent to all sites which have been identified in the sites list. The RC increments the **num_sites_sent** variable after each successful **LOCK_REQUEST**. If enough votes have been received from each of the sites is equal to the quorum, the RC sends a **COMPLETE** to the SGTM. If there is only one copy of the item on the system and it is local, the **lock_table** will be checked to determine whether to complete or abort the transaction. The procedure then appends the initiation time on the transaction which will be used in determining the transaction time out status.

do_remote_request(sender_name,trans_id,item_id)

This function is called from the **process_from_remote_RC()** procedure when a lock request is made from a local site to the remote sites. The function first calls the

lock_status() function to determine the transaction lock state. The **lock_status** function uses the **item_id** to find the locking status of the transaction from the **lock_table**. If the lock status is a (1), the **op_code** variable is set to **LOCK_REFUSED,** if the status is a (0), the **op_code** variable will be set to **LOCK_GRANTED**. To provide a measure of fault tolerance, if the locking status is not one of the two states already described, or the **item_id** does not exist at that site, the **lock_status** variable will default to **LOCK_REFUSED**. The outgoing message is then constructed and sent to the module port where it is sent to the remote sites using the SI library call **siSendto()**.

calculate_quorum(sender_name,op_code,trans_id,item_id)

This function is called from the **process_from_remote_RC()** procedure when **LOCK_GRANTED** or **LOCK_REFUSED** is received from a remote site. The function first navigates the **TRANSACTION** data structure queue to find the correct **trans_id**. After finding the correct **trans_id**, the **site_info** data structure is used to determine the number of votes (**s_votes**) for that site. If the **op_code** has been set to **LOCK_GRANTED** by the transaction, the total votes is added to the total by the number contributed by that site. If the total number of sites which have replied is equal to the total sites which sent messages and the number of votes received is not enough for a quorum, a **trans_abort** message will be sent back to the requesting site. If the total number of votes is equal to the read or write quorum number, a **TRANSACTION_COMPLETE** message is sent back to the requesting site. The transaction is also deleted from the transaction queue. If the total number of sites that responds is equal to the total number of sites which sent messages and an insufficient number of votes has been received, then a **trans_abort** message is sent to the SGTM. To handle a potential time-out situation where a site is waiting for a message, a timestamp is placed in the message to indicate when the message was initiated. If the last site has not replied in a period of 10 seconds, the transaction will be deleted from the queue and the **op_code** set to **TRANSACTION_ABORT** and sent to the local SGTM. This procedure will not work under all conditions unless a method such as a system timing message is implemented to increment the time-out checking. This would be necessary since the SI library function **siReceivefrom()** does not have provisions for handling time-out checking for individual messages.

delete_transaction(tid)

This function looks for the target transaction and manipulates pointers to delete the transaction from the queue.

lock_find(itemid)

This function looks up the item in the **lock_table** using the **itemid** to find the locking status of the transaction.

add_transaction(opcode,transid,itemid)

This function adds a transaction to the head of the transaction queue. The function first finds the replica data by using the **itemid**. The data structure for the transaction is then initialized with default values. The data structure for the **site_list** is then created for this transaction and the site name is obtained from the mapping table and

appended to the structure. As shown in Figure 2, the **site** and the **votes** for that site are then appended to the site information.

init_rc_datatables()

This procedure is called from the **main** procedure and is used to read in the transaction and locking data from files stored at the local site. The procedure first obtains the data file name from the local site. It then opens the replica data file and reads the transaction number, site number, and number of votes from each site into the transaction queue. The **compute_site_num()** function is called to calculate the number of sites contained in that transaction id. The **compute_quorum** function is then called to calculate the value of the read or write quorum. The **sitenamefile** file is then opened, and the values are read to initialize the **siteno_to_sitenme** table. The **Lockfile** is then opened and values of **itemid** and the **lock** (0 not locked, 1 locked) are read to initialize the **lock_table**.

skip(c,fp)

This function is used by the previous procedure to pass over various formatting (: ,) expressions which are contained in the files which are used to initialize the **locking** and **TRANSACTION** data structures.

compute_quorum(ptr1)

This function is used by the **init_rc_datatables()** procedure to calculate the read or write quorum for a given transaction. The for loop in the function sums the number of votes from each site which has replied to a read or write transaction request. The read quorum is then calculated by using the equation $\frac{v1 + v2 \ldots vn}{2} + 1$ and the write quorum is calculated by using the equation $\frac{v1 + v2 \ldots vn}{2} + 1$. The equations used to determine the quorums conform to the constraints mentioned in the Introduction for a voting algorithm.

compute_site_num(ptr1)

This function is used by the **init_rc_datatables()** procedure to calculate the number of sites which have a particular data item.

Operational Description

As shown in Figure 5, the operator first initializes the **SGTM** and **RC** modules at each site, and specifies socket numbers for both transmitting and receiving messages. Following initialization, the **RC** at each site enters a loop as a server where it waits for messages which occur at the receive port. When a message arrives, it is categorized as to its origin: whether it is from the local **SGTM** or from a remote **RC** which is located at another site(s). As described in the functional description section, the message is then parsed to determine its type (**LOCK_REQUEST**, **LOCK_GRANTED**, **LOCK_REFUSED**). Depending on the type of message received, either a read or write quorum is calculated, or a request **lock_request** message is constructed. If a **LOCK_GRANTED**, **LOCK_REFUSED** message is selected, the procedure deter-

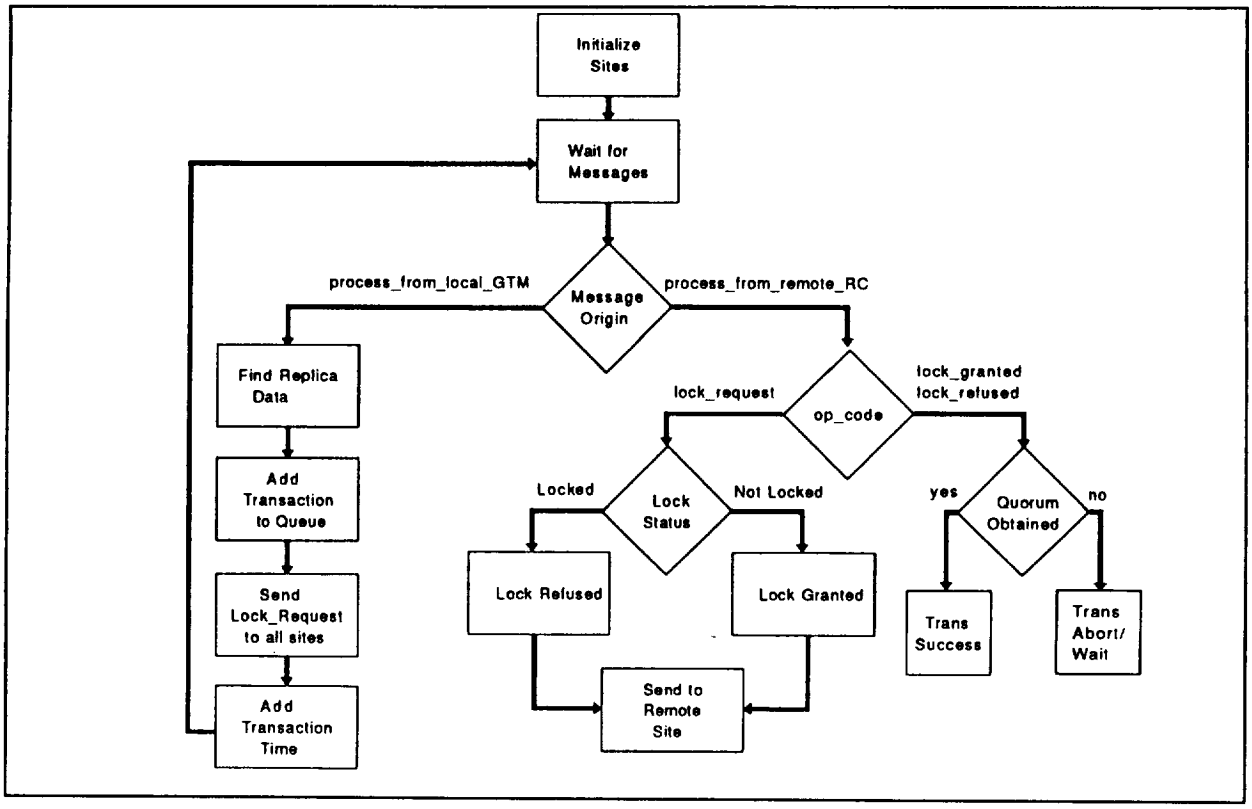


Figure 5. Replica Control Process

mines if a quorum is available. If the quorum is present, a success message is sent otherwise, the transaction is aborted, or if a site has not replied, the process will wait until the time-out period has elapsed.

Conclusion

This project has provided a unique perspective on the Replica Control problem as it relates to managing replicated data across a distributed network. The project has provided the following insights into this problem:

- A Replica Control algorithm must have sufficient performance and must provide adequate protection to the replicated data to prevent inconsistency in the data resident at that site.
- The algorithm must provide error detection and processing mechanisms which allow recovery after anomalous behavior by sites on the network.
- A weighted voting scheme is clearly more desirable than the single-vote majority consensus algorithm since it increases the overall probability that a quorum can be achieved under conditions where replicated data objects are limited. However, this method of handling replicated data is not as desirable as using *coteries*.[3]

- Even though *coteries* may be the most desirable method of replica control, they are much more complex to implement, and their performance suffers as the number of sets increases.

As was mentioned in the assumptions, transactions are given a time-stamp to give this RC implementation a degree of recovery from a site or gross communication link failure. In this project, no provisions were made for detecting corrupted messages and for accommodating dynamic site states in the distributed system.

References

1. S.B. Davidson, Replicated Data and Partition Failures, *Distributed Systems* edited by S. Mullender, ACM Press, New York, NY, 1989.

2. M. Raynal, Distributed Algorithms and Protocols, J. Wiley & Sons Ltd., 1988.

3. H. Garcia-Molina and D. Barbara, How to Assign Votes in a Distributed System, *J. of the ACM*, **32**(4): 841-860, Oct. 1985.

Appendix I

Replica Control

Project Source Code

```
#ifndef RC_H
#define RC_H

#include <stdio.h>
#include <math.h>
#include <time.h>
#include "/home/jiang_r/DS/src/common/si.h"

#define Read                    0
#define Write                   1
#define REPLIED                 1
#define NOT_REPLIED             0
#define COMPLETE                1
#define NOT_COMPLETE            0
#define LOCKED                  1
#define NOT_LOCKED              0

#define READ_REQUEST            0
#define WRITE_REQUEST           1
#define LOCK_REQUEST            2
#define LOCK_GRANTED            3
#define LOCK_REFUSED            4
#define TRANSACTION_ABORT       5
#define TRANSACTION_COMPLETE    6


#define MAXITEM                 50
#define MAXSITE                 10

struct tm *local_time;
time_t local_t;

char  *siteno_to_sitename[MAXSITE][80];

int   lock_table[MAXITEM][2];


/*  This structure is used with the logical_item structure and
stores the number of votes provided by a particular site which
possesses the data item.
*/

struct physical_item {
        int site;
        int vote;
        struct physical_item *phext;
};

/*  This structure provides specific information on a particular
site which possesses a copy of a data item requested in a
transaction.  It is an element of the TRANSACTION structure.
*/

struct site_info {
  char   sitename[80];
  int    sitelenth;
  int    status;
  int    s_vote;
  struct site_info  *next;
};

/*  This structure is a linked list element which stores
information on a specific data item within the database.
*/

struct logical_item {
        int item_ID;
        int r_quorum;
        int w_quorum;
        int site_num;
        struct physical_item *phirst;
        struct  logical_item *logext;
};

/*  This structure maintains all the required information of a
locally generated transaction request.
*/

typedef struct TRANSACTION TRANS;
struct TRANSACTION {
  int                   trans_id;
  int                   rw_quorum;
  int                   total_votes;
  int                   num_sites_sent;
  int                   num_sites_reply;
  int                   num_sites;  /* item has num_sites   */
                                    /* copies in the system */
  int                   send_time;
  struct site_info      *site_list;
  struct TRANSACTION    *next;
};


TRANS   *trans_queue_head;

/*  Maintains the pointer to the first data item within the
database.
*/

struct logical_item  *header;

#endif
```


1C714PRINT

```
/*-------------------------------------------------------------------------
| |                                                                       | |
  =>
| |  WEIGHTED VOTE BASED DISTRIBUTED REPLICA CONTROL PROGRAM              | |
  =>
| |                                                                       | |
| |-------------------------------------------------------------------------*/
#include "SRC.h"

char   modulehost[80];
char   moduleport[20];
SI     modulesi;

SI     *sdtoGTM;
SI     sdcome;
char   GTMport[20];
int    ier;

char   Replicafile[80];
char   Sitenamefile[80];
char   Lockfile[80];

/*************************************************************************/
/*                        FUNCTION DECLARATION                           */
/*************************************************************************/
void delete_transaction();
TRANS *find_transaction();
void add_transaction();
struct logical_item  *itemid_to_pointer();
void compute_site_num();
void compute_quorum();
void do_remote_request();
void calculate_quorum();
struct physical_item  *append_physical_item();
struct physical_item  *add_first_physical_item();
struct logical_item  *append_logical_item();
struct logical_item  *add_first_logical_item();
void init_rc_socket();
void init_rc_datatables();
int lock_find();
void process_from_remote_RC();
void process_from_local_GTM();
void skip();

/*************************************************************************/
/*                             MAIN PROGRAM                              */
/*************************************************************************/
main(argc, argv)

     int argc;
     char *argv[];
{

  char    *p;
  int     op_code;
  int     transaction_ID;
  int     trans_id;
  int     item_id;
  int     bytes;
  int     cc;
  char    buff[SIMSGSIZE];


  sprintf(moduleport, "%s", argv[1]);
  sprintf(GTMport, "%s", argv[2]);

  gethostname(modulehost, 80);
  siConvertName(modulehost, &ier);
  if(ier != SISUCCESSFUL ) {
    fprintf(stderr, "RC: failed on call to siconvertname\n");
    exit(10);
  }

  init_rc_datatables();
  init_rc_socket();

  while(1)
    {
      p =  siReceivefrom(&modulesi, &sdcome, &bytes, &ier);
      if (ier != SISUCCESSFUL) {
        fprintf (stderr, "Client GTM: failed on call to siReceive\n");
        continue;
      }

      if(!strcmp(sdcome.sender_port, moduleport))
        process_from_remote_RC(&sdcome, p, bytes);
      else
        if(!strcmp(sdcome.sender_port, GTMport))
          process_from_local_GTM(p, bytes);
        else printf("Error: Sender_port invalid.\n");
    }

}
/*************************************************************************/
/*                          END OF MAIN PROGRAM                          */
/*************************************************************************/
  =>


/*************************************************************************/
void process_from_remote_RC(sd, msg, bytes)
/*************************************************************************/
  =>
     SI *sd;
     char *msg;
     int bytes;

/* PURPOSE:  This procedure processes messages received from remote RC
modules.  These messages will either be requests for a read/write of a
locally held data item or replies from remote sites for a locally
initiated request to read/write a data item.*/

{
  int cnt;
  int op_code;
  int item_id;
  int trans_id;
  int lock;
  char  sender_name[80];


  /*   Get sender name  */
  strcpy(sender_name, sd->sender_machine);

  /* Parse the message into its fields */

  /* Get the op_code */
  cnt = 0;
  bcopy(&msg[cnt], &op_code, sizeof(int));

  if(op_code < 0) {  /*  1 */
```


1C14PRINT

```
    printf(" Op_code has a negative value.\n");
    exit(1);
  ) /* 1  */

  cnt += sizeof(int);

  /* Get the transaction id number  */

  bcopy(&msg[cnt], &trans_id, sizeof(int));

  if(trans_id < 0) {   /* 2  */
    printf(" Trans_id has a negative value.\n");
    exit(1);
  ) /* 2  */

  cnt += sizeof(int);

  /* Get the item id  */

  bcopy(&msg[cnt], &item_id, sizeof(int));

  if(item_id < 0) {   /* 3  */
    printf("Itemid has a negative value. \n");
    exit(3);
  ) /*  3 */

  /* Determine the message type and execute the respective function */

 switch ( op_code) {   /* 4  */

  case LOCK_REQUEST:  do_remote_request(sender_name, trans_id, item_id); break;
  case LOCK_GRANTED:  calculate_quorum(sender_name, op_code, trans_id, item_id);
  =>     break;
  case LOCK_REFUSED:  calculate_quorum(sender_name, op_code, trans_id, item_id);
  =>     break;
  default : printf("Invalid message received from %s\n", sender_name);
  ) /* 4 */

  return;
) /*  End of process_from_remote_RC function  */



/*************************************************************************/
void do_remote_request(sendername,trans_id,item_id)
/*************************************************************************/
  =>
     char  sendername[80];
     int * trans_id,item_id;

/* PURPOSE:  This function processes requests for read/writes from remote  sites
  => .*/
{
  int lock_status;
  int op_code;
  int cnt;
  char msg[SIMSGSIZE];
  SI   *sdtoRC;


 /* Find lock status of item_id */

  lock_status = lock_find(item_id);


  switch (lock_status) {   /* 1  */
  case LOCKED:     op_code = LOCK_REFUSED;
    break;

  case NOT_LOCKED: op_code = LOCK_GRANTED;
    break;

  default:
    op_code = LOCK_REFUSED;
    break;

  )   /* 1  */

  /*  Build the outgoing message  */

  cnt=0;
  bcopy(&op_code, &msg[cnt], sizeof(int));
  cnt += sizeof(int);
  bcopy(&trans_id, &msg[cnt], sizeof(int));
  cnt += sizeof(int);
  bcopy(&item_id, &msg[cnt], sizeof(int));
  cnt += sizeof(int);
  msg[cnt] = '\0';

  /* Send the message to the remote site  */

  sdtoRC = siSendtoInit(sendername,
                        moduleport, &ier);
  if (ier!= SISUCCESSFUL) {
    fprintf(stderr, "RC fail on call to remote siSendtoInit %d\n",sendername);
  =>  )


  siSendto(sdtoRC, msg, cnt, &modulesi, &ier);

  if (ier != SISUCCESSFUL) {   /* 2  */
    fprintf(stderr, "Client RC: failed on call to siSendto to remote site\n");
    exit(102);
  )   /* 2  */
  return;
)   /* End of do_remote_request function   */



/*************************************************************************/
void calculate_quorum (sendername, op_code,trans_id,item_id)
/*************************************************************************/
char  sendername[80];
     int  op_code,trans_id,item_id;

/* PURPOSE:  This function calculates the local quorum each time a grant  or ref
  => used message is received from
a remote site.*/
{
  struct site_info * site_ptr;
  int site_vote,new_opcode;
  int cnt;
  char msg[SIMSGSIZE];
  TRANS *temp_trans_queue_head;
  TRANS *temp_ptr;
  int timeout;
  int crntime;


  temp_ptr = (TRANS *)find_transaction(trans_id);

  if(temp_ptr == NULL)
```

```
    {
      /*  This means that the transaction has already been reported
      to the GTM, so just ignore it.  */
      return;
    }

  /* Execute only if the quorum is not complete   */
  /* Find the remote site name and vote info and update transaction structure */
  =>   site_ptr = temp_ptr->site_list;

  while(site_ptr) {    /*  2  */

    if(!strcmp(site_ptr->sitename, sendername)) {   /* 3  */

      site_ptr->status=REPLIED;
      site_vote=site_ptr->s_vote;
    }   /* 3  */

    site_ptr=site_ptr->next;
  }  /* 2 */

  if (op_code == LOCK_GRANTED) {   /*  4 */


    temp_ptr->num_sites_reply++;
    temp_ptr->total_votes= temp_ptr->total_votes + site_vote;

    /* Send trans abort msg to GTM if total sites which replied is equal to the
  => total sites sent msgs, but the number
of votes received is still not equal to the quorum required.
     */

 if( (temp_ptr->num_sites_reply == temp_ptr->num_sites_sent)  && (temp_ptr->tota
  => l_votes < temp_ptr
>rw_quorum) )
      {     /*  5  */


        new_opcode=TRANSACTION_ABORT;
        cnt=0;
        bcopy(&new_opcode, &msg[cnt], sizeof(int));
        cnt += sizeof(int);
        bcopy(&trans_id, &msg[cnt], sizeof(int));
        cnt += sizeof(int);
        bcopy(&item_id, &msg[cnt], sizeof(int));
        cnt += sizeof(int);
        msg[cnt] = '\0';

        /* Send the message to the GTM  */

        siSendto(sdtoGTM, msg, cnt, &modulesi, &ier);
        if (ier != SISUCCESSFUL) {   /* 5b */
          fprintf(stderr, "Client RC: failed on call to siSendto to remote site\
 => n");
          exit(102);
        }    /* 5b */

      }   /* 5  */
    /*  However, send trans complete msg to GTM if the total votes now
        equals the quorom.    */

    else {    /* 6  */

      if (temp_ptr->total_votes >= temp_ptr->rw_quorum)
        {    /* 7  */
```

```
  /*  Build the outgoing message  */

          /* Delete transaction from the queue */

          delete_transaction(trans_id);

          new_opcode=TRANSACTION_COMPLETE;
          cnt=0;
          bcopy(&new_opcode, &msg[cnt], sizeof(int));
          cnt += sizeof(int);
          bcopy(&trans_id, &msg[cnt], sizeof(int));
          cnt += sizeof(int);
          bcopy(&item_id, &msg[cnt], sizeof(int));
          cnt += sizeof(int);
          msg[cnt] = '\0';

          /* Send the message to the GTM  */

          siSendto(sdtoGTM, msg, cnt, &modulesi, &ier);
          if (ier != SISUCCESSFUL) {    /* 9  */
            fprintf(stderr, "Client RC: failed on call to siSendto to remote sit
  => e\n");
            exit(102);
          }   /* 9  */

        }   /* 7  */

    }   /*  6 */

  }   /*  4 */

  else {    /* 10 */

  if (op_code == LOCK_REFUSED) {    /* 11 */


      temp_ptr->num_sites_reply++;

      /* If the total number of sites responding now equals the total
         of sites sent messages, then send a trans abort message to
         the GTM.    */

      local_t = time(NULL);   /*  get current time  */
      local_time = localtime(&local_t);
      crntime = local_time->tm_sec;
      crntime += local_time->tm_min;
      crntime += local_time->tm_hour;
      crntime += local_time->tm_mday;

      timeout = crntime - temp_ptr->send_time;

      if ((temp_ptr->num_sites_reply == temp_ptr->num_sites_sent)
          || ( timeout > 10))
        {    /* 12*/

/*  Build the outgoing message  */

          /* Delete transaction from the queue */

          delete_transaction(trans_id);

          new_opcode=TRANSACTION_ABORT;
          cnt=0;
          bcopy(&new_opcode, &msg[cnt], sizeof(int));
          cnt += sizeof(int);
          bcopy(&trans_id, &msg[cnt], sizeof(int));
```

```
          cnt += sizeof(int);
          bcopy(&item_id, &msg[cnt], sizeof(int));
          cnt += sizeof(int);
          msg[cnt] = '\0';


          /* Send the message to the GTM  */

          siSendto(sdtoGTM, msg, cnt, &modulesi, &ier);
          if (ier != SISUCCESSFUL) {   /* 14 */
            fprintf(stderr, "Client RC: failed on call to siSendto to remote sit
   => e\n");
            exit(102);
          }    /* 14 */

        } /* 12 */

      }  /* 11  */

    }  /* 10  */

   return;
   /* End of calculate_quorum function  */


/*************************************************************************/
TRANS *find_transaction(tid)
/*************************************************************************/
/* PURPOSE:  Find the transaction within the queue and return the pointer  of th
  => e transaction record. If not found,
return a NULL pointer.*/
   int tid;
{
  TRANS  *trans_ptr;


  trans_ptr = trans_queue_head;

  while(trans_ptr != NULL)
    {
      if(trans_ptr->trans_id == tid)
        return (trans_ptr);
      trans_ptr = trans_ptr->next;
    }

  if(trans_ptr = NULL)
    return(NULL);

}  /* End of find_transaction()  */


/*************************************************************************/
void delete_transaction(tid)
/*************************************************************************/
  =>  int tid;

/*  PURPOSE: Delete the transaction from the transaction queue.          */
  =>

{
  TRANS  *trans_ptr;
  TRANS  *store_ptr;

  if(trans_queue_head->trans_id == tid)
    {
      trans_queue_head = trans_queue_head->next;
      return;  /* No transaction in the system  */
    }

  trans_ptr = trans_queue_head;
  store_ptr = trans_ptr;

/* Look for the transaction and remove it from the queue  */

  while(trans_ptr != NULL)
    {
      if(trans_ptr->trans_id == tid)
        {
          store_ptr->next = trans_ptr->next;
          return;
        }
      store_ptr = trans_ptr;
      trans_ptr = trans_ptr->next;
    }

  return;

} /*  End of delete_transaction()  */



/*************************************************************************/
void process_from_local_GTM(msg, bytes)
/*************************************************************************/
  =>    char *msg;
     int  bytes;
/*  PURPOSE:  This procedure initiates read or write qourum request messages   t
  => o remote sites once a transaction
request message is received from
the GTM. */

{
  int i = 0;
  int co = -1;
  int cnt = -1;
  int op_code;
  int operator;
  int itemid;
  int transid;
  int lock;
  char buf[SIMSGSIZE];
  struct logical_item *logical_ptr = NULL;
  struct physical_item *ph_ptr = NULL;
  int    sitelenth = 0;
  char   sitename[80];
  struct site_info  *site_ptr;
  SI   *sdtoRC;

 /*  Parse message into its specific fields   */

  co = 0;
  bcopy(&msg[co], &op_code, sizeof(int));
  co += sizeof(int);

  bcopy(&msg[co], &transid, sizeof(int));
  if(transid < 0) {  printf(" Trans_id has a negative value.\n");
                     exit(1);
                  }

  co += sizeof(int);
  bcopy(&msg[co], &itemid, sizeof(int));
  if(itemid < 0) {  printf("Item_id has a negative value.\n");
```

```
                exit(3);
              }


    /*  Find replica data for the itemid  */

    logical_ptr = itemid_to_pointer(itemid);

    if(logical_ptr == NULL)
      {
        /*  No such item in the system, so send a transaction abort message  */
    =>      /*  to the GTM.
                                    */
        printf("The data item does not exist in the database.\n");

        cnt = 0;
        operator = TRANSACTION_ABORT;
        bcopy(&operator, &buf[cnt], sizeof(int));
        cnt += sizeof(int);
        bcopy(&transid, &buf[cnt], sizeof(int));
        cnt += sizeof(int);
        bcopy(&itemid, &buf[cnt], sizeof(int));
        cnt += sizeof(int);
        buf[cnt] = '\0';

        siSendto(sdtoGTM, buf, cnt, &modulesi, &ier);
        if (ier != SISUCCESSFUL) {
          fprintf(stderr, "Client RC: failed on call to siSendto GTM\n");
          exit(102);
        }
        return;
      }

    /*  Add transaction to the head of the transaction queue  */
    add_transaction(op_code, transid, itemid);

  /* Construct LOCK_REQUEST message    */

    cnt = 0;
    op_code = LOCK_REQUEST;
    bcopy(&op_code, &buf[cnt], sizeof(int));
    cnt += sizeof(int);
    bcopy(&transid, &buf[cnt], sizeof(int));
    cnt += sizeof(int);
    bcopy(&itemid, &buf[cnt], sizeof(int));
    cnt += sizeof(int);
    buf[cnt] = '\0';

    /* Send a LOCK REQUEST message to all sites contained in the site list */
    site_ptr = trans_queue_head->site_list;
    while(site_ptr)
      {/*1*/
        /*  Determine if the site is remote or local  */

        if(strcmp(site_ptr->sitename, modulehost))
          {/*5*/
            /*   Send to remote RC  */
            sdtoRC = siSendtoInit(site_ptr->sitename,
                                  moduleport, &ier);
            if (ier!= SISUCCESSFUL) {
              fprintf(stderr, "RC fail on call to remote siSendtoInit %d\n",site_p
    => tr->sitename);
              }
            siSendto(sdtoRC, buf, cnt, &modulesi, &ier);
            if(ier != SISUCCESSFUL)  {
              fprintf(stderr, "RC fail on call send LOCK_REUEST to Remote RC\n");
              }
            else {
              /*  After message is sent, increase the num_site_sent  */
              trans_queue_head->num_sites_sent++;
            }
          }/*5*/
        else
          {/*6*/
            /*  Get the lock status of the local site  */

            /*  Look in lock_table for lock status     */
            lock = lock_find(itemid);
            if(lock == NOT_LOCKED)
              {/*3*/
                trans_queue_head->total_votes += site_ptr->s_vote;
                site_ptr->status = REPLIED;

                /*  Check if enough votes have been received  */
                if(trans_queue_head->total_votes >=
                   trans_queue_head->rw_quorum)
                  {
                    /*  Send to GTM a TRANSACTION_COMPLETE  message */
                    delete_transaction(transid);

                    cnt = 0;
                    operator = TRANSACTION_COMPLETE;
                    bcopy(&operator, &buf[cnt], sizeof(int));
                    cnt += sizeof(int);
                    bcopy(&transid, &buf[cnt], sizeof(int));
                    cnt += sizeof(int);
                    bcopy(&itemid, &buf[cnt], sizeof(int));
                    cnt += sizeof(int);
                    buf[cnt] = '\0';

                    siSendto(sdtoGTM, buf, cnt, &modulesi, &ier);
                    if (ier != SISUCCESSFUL) {
                    fprintf(stderr, "Client RC: failed on call to siSendto GTM\n")
    => ;
                    exit(102); }
                    return;

                  }
              }/*3*/

            if(lock == LOCKED)
              {/*4*/
                site_ptr->status = REPLIED;
                if(trans_queue_head->num_sites == 1)

                  {
    /*  If just one copy exists in the system and it is local send transaction_abor
     => t to GTM  */

                    delete_transaction(transid);
                    cnt = 0;
                    operator = TRANSACTION_ABORT;
                    bcopy(&operator, &buf[cnt], sizeof(int));
                    cnt += sizeof(int);
                    bcopy(&transid, &buf[cnt], sizeof(int));
                    cnt += sizeof(int);
                    bcopy(&itemid, &buf[cnt], sizeof(int));
                    cnt += sizeof(int);
                    buf[cnt] = '\0';
                    siSendto(sdtoGTM, buf, cnt, &modulesi, &ier);
                    if (ier != SISUCCESSFUL) {
                    fprintf(stderr, "Client RC: failed on call to siSendto GTM\n")
```

```
  => ;
                exit(102);
                }

                return;
              }
            }/*4*/
          }/*6*/
        site_ptr = site_ptr->next;  /*  Get next site_info  */

      }/*1*/

    /*   Determine the current system time and put into sending time  */
    local_t = time(NULL);
    local_time = localtime(&local_t);
    trans_queue_head->send_time = local_time->tm_sec;
    trans_queue_head->send_time += local_time->tm_min;
    trans_queue_head->send_time += local_time->tm_hour;
    trans_queue_head->send_time += local_time->tm_mday;


}/*end of process_from_local_GTM()  */

/*****************************************************************************/
int lock_find(itmid)
/*****************************************************************************/
     int itmid;

/*  PURPOSE:  Delete the transaction from the transaction queue.        */
  =>   {
  int i;

  i = 1;
  while(lock_table[i][0] >= 0)
    {
      if(itmid == lock_table[i][0])
      return(lock_table[i][1]);
      i++;
    }
  if(lock_table[i][0] < 0)
    return(-1);
}


/*****************************************************************************/
void add_transaction(opcode, transid, itemid)
/*****************************************************************************/
     int opcode;
     int transid;
     int itemid;

/*  PURPOSE:  Add transaction to the front of the tranaction queue.       */
  =>
{
  TRANS                   *temp_trans_ptr = NULL;
  struct logical_item     *logical_ptr = NULL;
  struct physical_item    *ph_ptr = NULL;
  struct site_info        *site_info_ptr = NULL;
  struct site_info        *store_info_ptr = NULL;
  char                    sname[80];
  int                     operator;
  char                    buf[SIMSGSIZE];
  int                     cnt;

  temp_trans_ptr = (TRANS *)calloc(1, sizeof(TRANS));

  /*  Find replica data for the itemid  */
  logical_ptr = itemid_to_pointer(itemid);

  /* Build the data structure  */
  temp_trans_ptr->trans_id = transid;
  temp_trans_ptr->total_votes = 0;
  temp_trans_ptr->num_sites_reply = 0;
  temp_trans_ptr->complete = NOT_COMPLETE;
  temp_trans_ptr->num_sites = logical_ptr->site_num;

  if(opcode == READ_REQUEST)
    temp_trans_ptr->rw_quorum = logical_ptr->r_quorum ;
  if(opcode == WRITE_REQUEST)
    temp_trans_ptr->rw_quorum = logical_ptr->w_quorum;

  ph_ptr = logical_ptr->phirst;

  /*  Create site_list for the transaction  */

  site_info_ptr = (struct site_info *)calloc(1, sizeof(struct
                                                        site_info));

  /*  Get site name from the mapping table  */
  strcpy(sname, siteno_to_sitename[ph_ptr->site]);
  strcpy(site_info_ptr->sitename, sname);
  site_info_ptr->sitelenth = strlen(sname);
  site_info_ptr->s_vote = ph_ptr->vote;
  site_info_ptr->status = NOT_COMPLETE;
  site_info_ptr->next = NULL;

  temp_trans_ptr->site_list = site_info_ptr;
  store_info_ptr = site_info_ptr;

  ph_ptr =ph_ptr->phext;

  while(ph_ptr != NULL)  /* Create site_list for the transaction  */
    {
      site_info_ptr = (struct site_info *)calloc(1, sizeof(struct site_info));
=>
      strcpy(sname, siteno_to_sitename[ph_ptr->site]);
      strcpy(site_info_ptr->sitename, sname);
      site_info_ptr->sitelenth = strlen(sname);
      site_info_ptr->s_vote = ph_ptr->vote;
      site_info_ptr->status = NOT_COMPLETE;
      store_info_ptr->next = site_info_ptr;
      store_info_ptr = site_info_ptr;
      ph_ptr = ph_ptr->phext;

    }

  /*  Insert into the transaction queue  */

  if(trans_queue_head == NULL)
    {
     /*  Insert as first element  */
      trans_queue_head = temp_trans_ptr;
    }
  else {
    temp_trans_ptr->next = trans_queue_head;
    trans_queue_head = temp_trans_ptr;
  }

} /* End of add_transaction()  */


/*****************************************************************************/
```

```
void init_rc_socket()
/*************************************************************************/
/*  PURPOSE: This procedure initiates the RC socket.                     */
  =>   {

  gethostname(modulehost, 80);
  siConvertName(modulehost, &ier);
  if(ier != SISUCCESSFUL ) {
    fprintf(stderr, "RC: failed on call to siconvertname\n");
    exit(10);
  }

  strcpy(modulesi.mymachine, modulehost);
  sprintf(modulesi.myport, "%s", moduleport);

  /* Initiate the RC with a correct port number*/
  siClientInit(&modulesi, &ier);
  if (ier != SISUCCESSFUL) {
    fprintf (stderr, "RC: failed on call to siClientInit\n");
    return;
  }

  /** get GTM  server information  ***/
  sdtoGTM = siSendtoInit(modulesi.mymachine, GTMport, &ier);
  if(ier != SISUCCESSFUL) {
    printf(" RC failed on call to siSendinit\n");
    return;
}

/*************************************************************************/
void init_rc_datatables()
/*************************************************************************/
{
  FILE *fp;
  int i, j;
  int itemid = -1;
  int site = -1;
  int vote = -1;
  int lock = -1;
  struct  logical_item *ptr1 = NULL;
  struct physical_item *ptr2 = NULL;


/*  PURPOSE:  This procedure contructs the local database from the database file
  => .*/

  /*  Get data file name  */

  strcpy(Replicafile, "/home/jiang_r/DS/src/PRJ/");
  strcat(Replicafile, modulehost);
  strcat(Replicafile, "_replicadata");
  strcpy(Sitenamefile, "/home/jiang_r/DS/src/PRJ/");
  strcat(Sitenamefile, modulehost);
  strcat(Sitenamefile, "_sitename");
  strcpy(Lockfile, "/home/jiang_r/DS/src/PRJ/");
  strcat(Lockfile, modulehost);
  strcat(Lockfile, "_lockdata");
  fp=fopen(Replicafile,"r");  /* Open the database file */
  if (fp==NULL) {
    fprintf(stderr, "\nError: Can't open %s .\n", Replicafile);
    exit(-1);
  }
  /*  Create the replica database table  */

  i=0;
  fscanf(fp, "%d", &itemid);
  while (itemid>0) {
    if (i==0)
      ptr1=add_first_logical_item(itemid);
    else
      ptr1=append_logical_item(itemid, ptr1);
    skip(':', fp);
    j=0;
    fscanf(fp, "%d", &site);
    while (site>0) {
      skip('/', fp);
      fscanf(fp, "%d", &vote);
      if (j==0)
        ptr2=add_first_physical_item(site, vote, ptr1);
      else
        ptr2=append_physical_item(site, vote, ptr2);
      j++;
      fscanf(fp, "%d", &site);
    }
    i++;
    fscanf(fp, "%d", &itemid);
  }
  fclose(fp);

  for (ptr1=header; ptr1!=NULL; ptr1=ptr1->logext) {
    compute_site_num(ptr1);
    compute_quorum(ptr1);
  }
  fp = fopen(Sitenamefile, "r");  /*  Initiate siteno_to_sitename table  */
  if(fp  == NULL) {
    fprintf(stderr, "\nError: Can't open file %s .\n", Sitenamefile);
    exit(-1);
  }
  i = 1;
  fscanf(fp, "%s", siteno_to_sitename[i]);
  while(strcmp(siteno_to_sitename[i], "-1"))
    {
      i++;
      fscanf(fp, "%s", siteno_to_sitename[i]);
    }
  fclose(fp);

  fp = fopen(Lockfile, "r");  /*  Init locking table  */
  if(fp  == NULL) {
    fprintf(stderr, "\nError:  Can't open file %s .\n", Lockfile);
    exit(-1);
  }

  i = 0;
  fscanf(fp, "%d:%d", &itemid, &lock);
  while(itemid >= 0)
    {
      i++;
      lock_table[i][0] = itemid;
      lock_table[i][1] = lock;
      fscanf(fp, "%d:%d", &itemid, &lock);
    }
  lock_table[i+1][0] = -1;  /*  This means EOF of lock_table  */
  fclose(fp);

} /*  End of init_data()  */


/*************************************************************************/
void skip(c, fp)
/*************************************************************************/
  =>    char c;
```

```
    FILE *fp;

/*  PURPOSE:  Used during reading of data files to aid database structure creati
  => on operations.*/
{
  while(c!=getc(fp));
}

/*************************************************************************/
struct logical_item * add_first_logical_item(itemid)
/*************************************************************************/
int itemid;

/*  PURPOSE:  This procedure adds the first data item to the database. */
{
  struct logical_item *ptr1 = NULL;
  ptr1=(struct logical_item *)calloc(1, sizeof(struct logical_item));

  ptr1->item_ID=itemid;
  ptr1->phirst=NULL;
  ptr1->logext=NULL;
  header=ptr1;

  return(ptr1);
}

/*************************************************************************/
struct logical_item * append_logical_item(itemid,ptr)
/*************************************************************************/
int itemid;
struct logical_item *ptr;

/*  PURPOSE: This procedure adds subsequent data items to the database.  */
  =>   {
  struct logical_item *ptr1 = NULL;
  ptr1=(struct logical_item *)calloc(1, sizeof(struct logical_item));

  ptr1->item_ID=itemid;
  ptr1->phirst=NULL;
  ptr1->logext=NULL;
  ptr->logext=ptr1;

  return(ptr1);
}

/*************************************************************************/
struct physical_item * add_first_physical_item(site, vote, ptr1)
/*************************************************************************/
  =>    int site;
    int vote;
    struct logical_item *ptr1;

/*  PURPOSE: This procedure adds the first site which possesses a
particular database item to the site list for this database item.  */
{
  struct physical_item *ptr2 = NULL;

  ptr2=(struct physical_item *)calloc(1, sizeof(struct physical_item));

  ptr2->site=site;
  ptr2->vote=vote;
  ptr2->phext=NULL;

  ptr1->phirst=ptr2;

  return(ptr2);
}

/*************************************************************************/
struct physical_item * append_physical_item(site, vote, ptr)
/*************************************************************************/
int site;
int vote;
struct physical_item *ptr;

/*  PURPOSE: This procedure adds subsequent sites to a database item's site list
  =>  information structure.*/
{
  struct physical_item *ptr2 = NULL;

  ptr2=(struct physical_item *)calloc(1, sizeof(struct physical_item));
  ptr2->site=site;
  ptr2->vote=vote;
  ptr2->phext=NULL;
  ptr->phext=ptr2;

  return(ptr2);
}

/*************************************************************************/
void compute_quorum(ptr1)
/*************************************************************************/
struct logical_item * ptr1;

/*  PURPOSE:  This procedure computes the read and write quorums required   for
  => a particular database item based
on the total number of votes.  The  calculated quorum is weighted and holds true
  =>  for the following equations:
 1) Total votes of read and write qourums > total # of votes
 (2) Total write votes > total # of possible votes/2
The equations in this procedure which calculate the required read and write quor
  => ums do not have a particular basis
but hold true to the above constraints. */
{
  struct physical_item * ptr2 = NULL;
  int    sum = 0;
  int    count = 0;

  sum=0;
  count = 0;
  for (ptr2=ptr1->phirst; ptr2!=NULL; ptr2=ptr2->phext) {
    sum += ptr2->vote;
    count ++;
  }

/*  The read and write quorums are calculated as follows:*/
  ptr1->r_quorum = sum/2 + 1;
  ptr1->w_quorum = sum/2 + 1;
}

/*************************************************************************/
void compute_site_num(ptr1)
/*************************************************************************/
struct logical_item * ptr1;

/*  PURPOSE:  This procedure determines the total number of sites which  possess
  =>  a particular database item.*/

{
  struct physical_item * ptr2 = NULL;
  int    sum = 0;
```

```
  sum=0;
  for (ptr2=ptr1->phirst; ptr2!=NULL; ptr2=ptr2->phext) {
    sum++;
  }

  ptr1->site_num=sum;
}

/*************************************************************************/
struct logical_item * itemid_to_pointer(itemid)
/*************************************************************************/
/*  PURPOSE:  This procedure returns a pointer to the required database item. */

{
  struct logical_item *ptr1 = NULL;
ptr1=header;
  while (ptr1!=NULL && ptr1->item_ID!=itemid) ptr1=ptr1->logext;
  return(ptr1);
}

/*************************************************************************/
/*                          END OF PROGRAM FILE                          */
/*************************************************************************/
```

AN INTERACTIVE USER INTERFACE FOR A DISTRIBUTED SYSTEM ON N NODES

by
Yagnamurthy Sekhar
Shailesh Rao
Gary Carlson

TEAM 1
DISTRIBUTED SYSTEMS PROJECT
CS 763
10 December 1992

GOALS — The goals of this team's programming effort are to implement a program which will automatically establish a distributed system (DS), based upon parameters which are input by the initiating user, AND allow the user to interact with this DS in real time.

PARAMETERS — Ideally, some of the PARAMETERS which a user may wish to input might include: {N.B. not all of the below will be available in this implem.}

names of sites to be included in the DS {node list}

algorithm or program to be run after network is established

network topology desired

verbose or non–verbose reporting; report to a logfile

connection type

OBJECTIVES — The programming team (Team) chose to list the OBJECTIVES in order of achievability. This list will serve as objectives for future efforts and inspiration for further thinking or research into this aspect of programming for DS. Some of the means of implementation are indicated within the braces; a brief summary of mechanisms and operations follow in the section labelled "SUMMARY," as well as in the comments within the source code. A detailed description of the implementation is provided in the section labelled "DETAILS."

Choose a programming language {C}

Define the important parameters {see above}

Choose the exact type of connection {sockets}

Choose network topology for this project implementation {logical ring}

Choose an algorithm for this project implementation {token passing}

Define the format of the message {see DETAILS}

Build the network of nodes {see DETAILS}

Begin circulating messages in accordance with the chosen protocol

Allow user to change certain parameters during execution {not implem}

SUMMARY

Message Format — Token_bit(int 0|1); source(string, e.g. horus.cs.odu.edu); destination(string, e.g. lilac.cs.odu.edu); data(string); ack(int 0|1); msg_id(int). The strings are delimited by \n, allowing them to vary in length.

Connection type — Sockets are used in this implementation as the means of communication. At present during network activation, the Pilot "remote execs" the server program on the successor node and then listens on its in_port for 5 minutes. If nothing is heard, Pilot passes the NULL string and then terminates itself. In a real system, some mechanism other than a timeout and self–termination would be desirable to cope with failure of the completion of the ring.

Algorithm choice — Specify the algorithm to be run as .c unless only .o available, in which case we need to know machine and architecture types. The Team has chosen to design, code, and implement a general purpose token passing program to exercise the established network. When the algorithm is called, whatever it is, the following will be passed as arguments:

list of active nodes on Distributed System
Socket descriptors to be used for message passing
Algorithm to be used or address where its code may be found

Termination — System termination is presently effected by passing a NULL string to one's successor in the ring. If string == NULL, the node passes the NULL string to its successor and then terminates itself.

DETAILS — This section describes the means by which our more involved objectives were achieved. Included in such descriptions will be the problems encountered or discussed, the resolution decided upon with justification, and the procedure by which the resolution was effected.

Connection type — TCP sockets are used for communication between nodes owing to its reliabilty and intrinsic suitability for this distributed system.

Establish the network topology and protocol — Naturally, the system must be activated on a single node, the one on which the user is logged on. We call this node the "pilot," and label it P0. (See Figure 1.) Once executed, the program functions as follows:

Explanation of PILOT & SERVER portions of program

The distributed system with INTERACTIVE USER INTERFACE developed by Team 1 to complete the course project consists of two parts. Detailed descriptions of these two parts will follow with their respective pseudocode portions. The source code for the system is in two files pilot.c and server.c.
-- The first part is a PILOT program which

(1) takes input from the user,

(2) starts the entire process and

(3) calls the application algorithm. In our implementation, this is the token ring protocol. PILOT initializes the ring protocol by pumping a token into the ring when it has confirmed that the ring is properly established.

-- The second part is a SERVER program which establishes and maintains connections with its predecessor and successor.

---

pilot.c:

The user of the distributed ring system runs the executable code of the file PILOT from a terminal. The program starts by requesting that the user input a list of the machines to be included in the network. Thus the PILOT program has the information of the members of the system. By default the machine from which PILOT is started is included in the system.

The PILOT program creates a socket to which its successor node can connect as part of the ring. The PILOT then forks a process which remotely executes the SERVER program on the successor node, found on the list which was input by the user.

The port number of the out_socket created by the PILOT is passed as an argument to the successor node. That node also needs the name of the

machine to which it should connect, hence this is also a parameter to be passed to the remotely executed "SERVER" process as an argument. The final argument to the remote execution is the port number of the in_socket of the PILOT. This is neccessary because the PILOT program, after starting its successor, waits on an in_socket for the last member of the ring to get connected to the PILOT, thus completing the ring. So the last node must know the port number of the socket on which the PILOT is ready to accept connection. This port number is therefore passed as a parameter to successive SERVER programs but only the last node uses this information; all other processes just pass it on to the next process. Once the ring is completed, the PILOT process forks a child which pumps the token into the ring and then 'exec's the program written for simulating the token ring protocol.

```
begin
        take list of nodes from user;
        create two sockets: in_sd, out_sd;
        bind both sockets to arbitrary ports;
        fork();
        if (child)
          rsh server process on next machine
          pass in_port, out_port and localhost name as
          parameters;

        if(parent)
          wait on out_sd for successor to get connected;
          pass the list of machines to be in the ring;

          wait on in_sd for the last member to get connected;

          /* ring is completed*/

          fork();

          if(parent)
                pause(); /* parent can be assigned any future
                            work*/

          if(child)
                put the token onto the ring through out_sd;
                exec the token ring program and pass to the
                  program in_sd, out_sd, list of members as
                  parameters;
```

end.

---

server.c:

The SERVER process gets as parameters three values:

* pre_port, the port on which its predecessor is waiting for connection;
* pilot_port, the port number on which PILOT will be expecting connection from the last member of the ring, so this value is to be passed on to the last member of the ring; and
* pre_host, the name of the predecessor.

By using the pre_host and pre_port parameters, the SERVER process gets itself connected to its predecessor and waits to receive the list of members of the ring. Once it gets the list it looks to see its own position in the ring. If it is not the last member, it gets the name of its successor node from the list. It then creates an 'out_sd' socket for the successor node to connect to. Then it forks a child process which remote starts the SERVER process on the successor node and passes to it as parameters the port number of out_sd, pilot_port and its own name. Parent waits on out_sd for the successor node. Once successor node is connected it passes the list of members of the ring which it got from its predecessor and it then 'exec's the token ring simulation process. When the last member of the ring when gets the list from its predecessor and finds out that it is the last node, it does not remote execute SERVER process again. Instead, it knows that PILOT is waiting on pilot_port for completion of the ring. It simply connects to the PILOT and then itself 'exec's the token ring simulation process.

Pseudocode for the SERVER process is as follows:

```
begin
        get the pre_port, pilot_port, pre_host as parameters;
        create in_sd socket and connect to pre_host at pre_port;
        receive list of members of the ring;
        open out_sd socket and bind it to a port;
        find out own position in the list;

        if(!last member)
         {          get the nexthost name;
          fork();
```

```
        if(child)
                remote start server on nexthost, pass out_sd port
                  number, pilot_port, localhost name as
                  parameters;
        if(parent)
                wait for connection on out_sd;
                pass list of members to successor;
                exec token ring simulation process and pass as
                  arguments in_sd, out_sd, list of members.
        }
        if(last member)
        {
                get PILOT host name from the list;
                connect to PILOT host on pilot_port;
                exec token ring simulation process and pass as
                  arguments in_sd, out_sd, list of members.
        }                                                        end.
```

The details for the TOKEN RING portion of the program are as follows:

The Token Ring program that is implemented as the application being run on the distributed system is a simple simulator that emulates the token ring protocol. The token format used is a character string with different fields, each performing a particular function. We use a template frame that serves both as a message and a frame. The first integer in the template is a 0 if it is a frame and a 1 if it is a token. The program works as follows:

The program waits on the input socket till it receives a message. A message in this program can only be one of two types : either the template string described above or a kill signal. In the case of the latter, the program terminates. If it receives the former then :

It checks the first field (which is read as an integer) and if it is a 1 then :

The message is a token . In which case the program decides whether to send a message on the outsocket or not. This is decided based on a randomly generated probability (if > 0.5, then send). If the decision to send a message is made, the program randomly decides the destination, picking a number from the list of nodes on the ring. Eventually, it sends a message (the data field of the message is fixed in this program for simplicity) and logs the message on its log

file by giving it a unique message id . It then converts the first integer to a 0 and sends out the template as a token. This way the token is kept circulating on the ring.

If the first field of the message was a 0 then the program knows that this is a message as opposed to a token. In this case, the program checks the destination field and compares it with its own name field. If they match , this implies that the message is destined for the machine in question. The program logs the fact that it received the particular message and sets the ack bit to a 1 (on the same template message string) and sends it back out on the ring. (source removal policy followed) .

If the destination field was not its own , then it checks to see if the source field is its own, which implies the message has come back to itself after having been received by its destination and hopefully, having been acknowledged. If this is true, the program logs the message and discards it.

If none of these cases is true, it means the message is just circulating and is destined for someone else. So the program just sends it out on its out_socket.

```
==================================================
Begin message passing
   create and release a token;        /* Pilot does this */
   for each node
      if token_present {
      use pseudo-random number generator (prng) to decide
         whether to send message or pass token;
         if (decision == send_msg) {
            use prng to select a destination node;
            send_msg;
            ciculate token
         }  /* endif decision == send_msg */
      else forward_token;
      }       /* endif token_present */

      if rcv_msg {
         if (msg_for_me) {
            process_msg;
            send_ack; }          /* endif msg_for_me */
      else if (source == self)
         remove_msg;
```

```
else pass_msg;
}       /* endif rcv_msg */
```

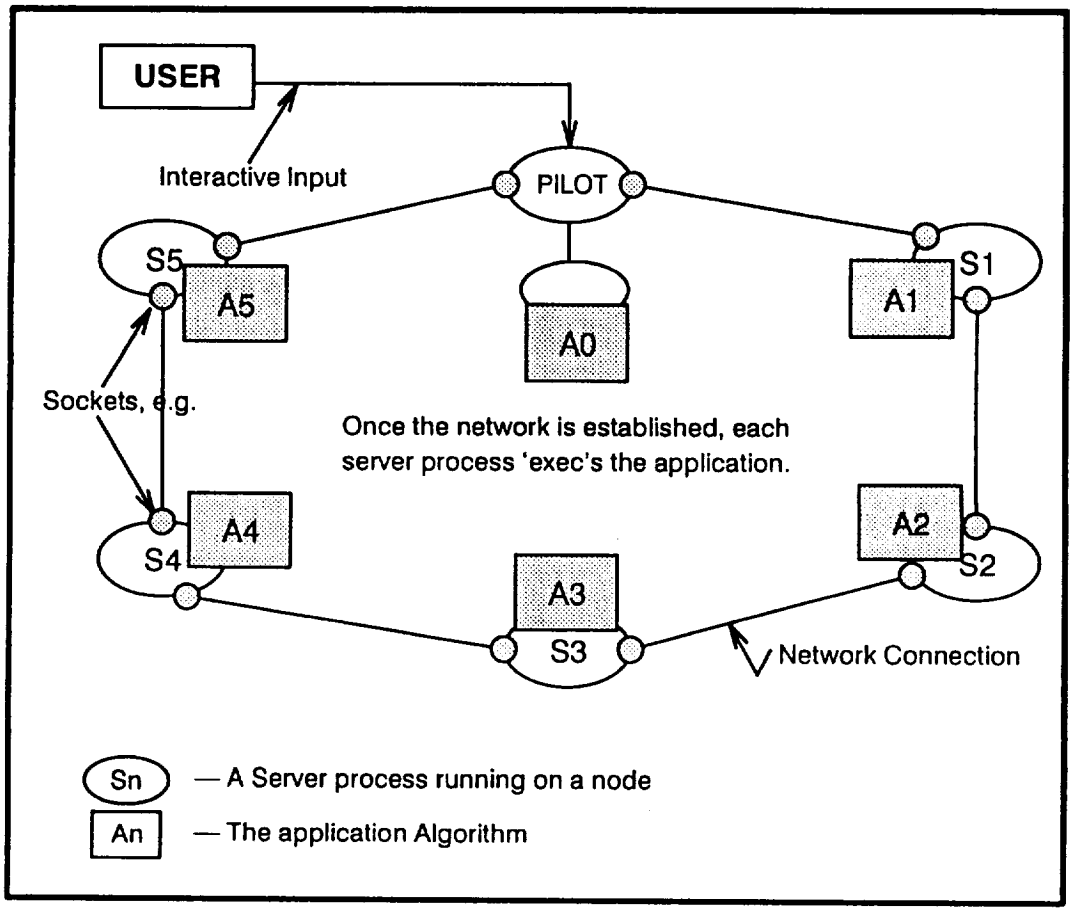


Figure 1. **Network Topology**

FUTURE ENHANCEMENTS — These are "wish–list" as well as "failure–proofing" items which were impractical to implement within the time frame of this semester. They can be used as ideas for future projects and as springboards for further thinking on the subject.

— If implemented in Xwindows, parameters could be chosen from pull–down lists.

— Store parameters in a formatted file which would be read by the "pilot" process. This would relieve the user from typing them in, yet the parameters would not be hard–coded into the program.

— Add or remove nodes at will during execution of the program

— Ping a node before attempting to establish a connection. If not alive, remove the inactive node's name from the node_list; continue with

successor node in node_list. When network is established and Pilot has received the node_list once again, it compares this received list to the original which was circulated. Report number of currently participating nodes and names of inactive nodes to the user; query the user whether they wish to add substitute nodes or continue. If additional nodes were specified, these would be added to the network, and a final node_list, superseding the original, would be circulated to all so that every node will have an identical list from which to pick its nodes.

— Maintain a complete logfile of all network activity. This might ultimately prove to be a burdensome overhead, clogging the network with reporting messages. However, it could be useful for debugging and/or analysis on a small scale program.

— Build a user–specified topology rather than only a ring. By making the "Pilot" program more intelligent, it could create the topology specified by the user.

```
/*******    PILOT.C *****************
This is the pilot program for establishing the ring topology with the
machines specified by the user:
************************************/

#include <stdio.h>
#include <errno.h>
#include <sys/time.h>

#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <sys/uio.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>

#define MAXHOSTNAME 80
#define MAXLEN  1024

int    i, out_sd, in_sd;

struct host_names{
  char   machine[80];
  struct host_names *next;
}

/*-------------------------------------------------------------------------*/
main(){
  struct host_names *hosts = NULL, *temp;
  char   *buffer, argv[20][40], buf[40], *tmp_buf;
  char localhost[80], command[80], next_host[80],
       other_host[80], token[MAXLEN];
  struct   sockaddr_in server1, server2, client1, client2, from;
  struct  hostent *hp, *gethostbyname();
  struct iovec *iov;
  int    num_machines, handle_alarm(), psd,
         fromlen, child, rex_child = 111, num;

  /*get the name of the local host */

  gethostname(localhost, MAXHOSTNAME);

  /*allocate space for buffer*/

  buffer = (char *)calloc(MAXLEN, sizeof(char));

  /*fill in the server1 structure for binding the outgoing socket*/

  server1.sin_family = AF_INET;
  server1.sin_port = htons(0);

  if((out_sd = socket (AF_INET, SOCK_STREAM, 0)) < 0){
    perror("creating socket");
    exit(200);
  }

  if  ( (hp = gethostbyname(localhost)) == NULL ) {
    perror("Can't find host %s\n", localhost);
    exit(-1);
  }
  bcopy ( hp->h_addr, &(server1.sin_addr), hp->h_length);
```

```
    /*bind the socket so that successor can get connected */

    if (bind( out_sd, &server1, sizeof(server1) ) < 0){
      perror("requested port is busy");
      close(out_sd);
      exit(300);
    }

    /* now work on binding the in socket for the last guy to complete */
    /* the ring  by getting connected back to me*/

    server2.sin_family = AF_INET;
    server2.sin_port = htons(0);
    bcopy ( hp->h_addr, &(server2.sin_addr), hp->h_length);

    if((in_sd = socket (AF_INET, SOCK_STREAM, 0)) < 0){
      perror("creating socket");
      exit(100);
    }

    if (bind( in_sd, &server2, sizeof(server2) ) < 0){
      perror("requested port is busy");
      close(in_sd);
      exit(300);
    }

    fromlen = sizeof(from);

    /* create space for holding machine names to be given by user*/

    hosts = (struct host_names *)malloc(sizeof(struct host_names));
    temp = hosts;

    /*The machine on which this process is run is invariably a member of */
    /*the ring topology so let me put my name in the list*/

    strcpy(temp->machine, hp->h_name);

    printf("Give the names of machines to be connected in a ring:\n");
    printf("ONE MACHINE NAME PER LINE PLEASE\n");

    /* Take the list of machines to be connected, RING is the default */
    /* topology forced for now. It can be modified*/

    while((scanf("%s", other_host)) != EOF){
      temp->next = (struct host_names *)malloc(sizeof(struct
                                                host_names));

      if  ( (hp = gethostbyname(other_host)) == NULL ) {
        printf("Can't find host %s\n", other_host);
        exit(-1);
      }

      /* for the sake of uniformity, convert the given name into "dot" */
      /* notation like 'offa.cs.odu.edu' */

      strcpy(temp->next->machine, hp->h_name);
      temp = temp->next;
    }
    temp->next = hosts;
    hosts = hosts->next;
    temp->next->next = NULL;

    /* get the port numbers to which two sockets are bound*/

    if (getsockname(in_sd,&client1,&fromlen)<0){ /*get client socket info*/
```

```
    perror("could't get sockname\n");
    exit(10);
  }

  if (getsockname(out_sd,&client2,&fromlen)<0){ /*get client socket info*/
    perror("could't get sockname\n");
    exit(10);
  }

  /* prepare the command to be executed by the child. */

  sprintf(command, "rsh -n %s /home/yagna_s/cs763/project/server %hu %hu %s",
          hosts->machine, ntohs(client2.sin_port),
          ntohs(client1.sin_port), localhost);

  /*now fork a child and let the child execute the rsh command because */
  /*it is a blocking call which won't return until the remote process */
  /*is terminated*/

  if((rex_child = fork()) < 0){
    perror("problem with forking the rex_child");
    exit(50);
  }

  else if(rex_child == 0){                    /* child process  */
    system(command);
    exit(55);
  }

  else{                                      /* parent again */

    listen(out_sd,1);     /* listen for connection from successor */

    out_sd  = accept(out_sd, &from, &fromlen);

    printf("%s: Establishing Ring.........Please wait.....\n", localhost);

    temp = hosts;
    tmp_buf = buffer;

    /* prepare the string of all machines of the ring to be passed to */
    /* the neighbor*/

    while(temp){
      sprintf(tmp_buf, temp->machine);
      tmp_buf += strlen(temp->machine);
      tmp_buf[0] = '\n'; tmp_buf++;
      temp = temp->next;
      num_machines++;
    }

    /* send the list of machines */

    if(send(out_sd, buffer, MAXLEN, 0) < MAXLEN){
      perror("sending on socket");
      exit(112);
    }

    alarm(300);

    /* now I am done. I wait for completion of ring by the last node */
    /* in the ring */

    listen(in_sd,1);
    in_sd  = accept(in_sd, &from, &fromlen);
```

```
        /*Oh!! Ring is completed. Let me turn off the alarm*/

        alarm(0);

        hp = gethostbyaddr(from.sin_addr);
        printf("%s:Oh!!!!!!Ring is completed from %s\n", localhost, hp->h_name);

        printf("Now I start the tkn_ring process\n");

        /* Initialize the token to start the process*/
        /*this is the format of the token agreed to by myself and */
        /*tkn_ring*/

        strcpy(token, "1\nhorsa.cs.odu.edu\noffa.cs.odu.edu\nNOTE\n0\n1\n");
        send(out_sd, token, MAXLEN);

        /*Now let me take rest and let my child run the tkn_ring process*/

        if((child = fork()) < 0){
          perror("problem with forking the child");
          write(out_sd, "", strlen(""));
          exit(400);
        }
        else if(child > 0){     /*parent process*/
          pause();
        }
        else{                   /* here is my child */
          char in_socket[10], out_socket[10];

          sprintf(in_socket, "%d", in_sd);
          sprintf(out_socket, "%d", out_sd);
          strcpy(argv[0], "tkn_ring");
          sprintf(argv[1], in_socket);
          sprintf(argv[2], out_socket);

          /* pass in_sd, out_sd and list of members in the ring to the */
          /* process */

          execlp("tkn_ring", argv[0], argv[1], argv[2], buffer, (char *)0);
        }
      }
    }
    /*-------------------------------------------------------------------*/

    handle_alarm()
    {
      printf("Some thing is wrong, timed out for ring establishment\n");

      write(out_sd, "", strlen(""));
      exit(500);
    }

    /*-------------------------------------------------------------------*/
```

```
/********** SERVER.C ************************
This is the server program executed by the pilot program on all the
machines which are to be part of the ring
*************************************************/

#include <stdio.h>
#include <errno.h>
#include <sys/time.h>

#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <sys/uio.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>

#define MAXHOSTNAME 80
#define MAXLEN  1024

int    i, out_sd, in_sd;

struct host_names{
  char   machine[80];
  struct host_names *next;
}
/*-------------------------------------------------------------------------*/
main(argc, argsv)
     int argc;
     char **argsv;
{
  struct host_names *hosts = NULL, *temp;
  char   *buffer, argv[20][40], buf[40], nodes[10][40], *tmp_buf;
  char localhost[80], command[80], next_host[80],
       other_host[80], token[10], pre_host[80];
  struct   sockaddr_in server1, server2, client1, client2, from;
  struct  hostent *hp, *gethostbyname();
  int    num_machines, handle_alarm(), psd,
         fromlen, child, rex_child = 111, num,
         pre_port, pilot_port;

  /* server must receive three arguments from pilot and fourth one is */
  /* from the shell which is the name of the program*/

  if(argc < 4){
    perror("Not enough arguments are passed to the server:");
    exit(10);
  }

  pre_port   = atoi(argsv[1]);     /*predecessor port I must connect to */
  pilot_port = atoi(argsv[2]);  /*in_port of pilot, used by last member*/
  strcpy(pre_host, argsv[3]);     /* predecessor's name*/

  /* let me find my own name first */

  gethostname(localhost, MAXHOSTNAME);

  buffer = (char *)calloc(MAXLEN, sizeof(char));

  /* now I will open a socket for connecting to my predecessor */

  server1.sin_family = AF_INET;
  server1.sin_port = pre_port;
```

```
if((in_sd = socket (AF_INET, SOCK_STREAM, 0)) < 0){
  perror("creating socket");
  exit(200);
}

if  ( (hp = gethostbyname(pre_host)) == NULL ) {
  perror("Can't find previous host");
  exit(-1);
}
bcopy ( hp->h_addr, &(server1.sin_addr), hp->h_length);

/* I will create a socket for my successor as well */

server2.sin_family = AF_INET;
server2.sin_port = htons(0);

if((out_sd = socket (AF_INET, SOCK_STREAM, 0)) < 0){
  perror("creating socket");
  exit(200);
}

if  ( (hp = gethostbyname(localhost)) == NULL ) {
  perror("Can't find local host");
  exit(-1);
}
bcopy ( hp->h_addr, &(server2.sin_addr), hp->h_length);


if (bind( out_sd, &server2, sizeof(server2) ) < 0){
  perror("requested port is busy");
  close(out_sd);
  close(in_sd);
  exit(300);
}

/*sleep for a while until my predecessor executes accept()*/

sleep(2);

if ( connect(in_sd, &server1, sizeof(server1)) < 0 ) {
  close(in_sd);
  close(out_sd);
  perror("connecting stream socket");
  exit(150);
}

/* I am part of the ring now!!!!!!!!*/

fromlen = sizeof(from);

if (getsockname(in_sd,&client2,&fromlen)<0){ /*get client socket info*/
  perror("could't get sockname\n");
  exit(10);
}

sleep(2);

if(recv(in_sd, buffer, MAXLEN, 0) < 0){
  perror("receiving on in_socket for server");
  exit(11);
}

/* Now I got the list of nodes to be in the ring. Thus I know my */
/* successor also */
```

```
    tmp_buf = buffer;
    i = 0;

    /* let me sort out the list of nodes to start with */

    while(sscanf(tmp_buf, "%s", buf) >0){
      strcpy(nodes[i++], buf);
      tmp_buf += strlen(buf) + 1;
    }
    strcpy(nodes[i], "");

    /* I should know my position in the ring, whether I am the last */
    /* one*/

    for(i=0; strcmp(nodes[i], ""); i++)
      if(!(strcmp(hp->h_name, nodes[i])))     /*hp is still the localhost*/
        break;
    strcpy(next_host, nodes[++i]);

    if(strcmp(nodes[++i], "")){          /* I am not the last one. So let */
                                         /* the other guys also enter ring*/

      if (getsockname(out_sd,&client1,&fromlen)<0){ /*get client socket info*/
        perror("could't get sockname\n");
        exit(10);
      }

      sprintf(command, "rsh %s /home/yagna_s/cs763/project/server %d %d %s",
              next_host, client1.sin_port, pilot_port, localhost);

      /* I will let my child create the next guy in the ring */

      if((rex_child = fork()) < 0){
        perror("forking rex_child");
        exit(350);
      }

      else if(rex_child == 0){
        system(command);
        exit(1);
      }

      else{                  /* I am back again with business */

        listen(out_sd,1);
        if((out_sd  = accept(out_sd, &from, &fromlen)) < 0){
          perror("accepting connection");
          exit(13);
        }

        /* Next node is connected so, pass the list of nodes to him*/

        if(send(out_sd, buffer, MAXLEN) < MAXLEN){
          perror("sending buffer");
          exit(14);
        }
      }
    }
    else{                  /*I am the last node, so connect back to the */
                           /*pilot to complete the ring*/

      server1.sin_port = pilot_port;

      if(!(hp = gethostbyname(next_host))){
        perror("pilot host is not found");
        exit(111);
```

```
        }
        bcopy(hp->h_addr, &(server1.sin_addr), hp->h_length);

        printf("connecting to pilot");

        sleep(2);

        if ( connect(out_sd, &server1, sizeof(server1)) < 0 ) {
          close(in_sd);
          close(out_sd);
          perror("connecting to pilot socket");
          exit(150);
        }
      }

      /*now let me start the actual exhibition of tkn_ring process*/

      {
        char in_socket[10], out_socket[10];

        sprintf(in_socket, "%d", in_sd);
        sprintf(out_socket, "%d", out_sd);
        strcpy(argv[0], "tkn_ring");
        sprintf(argv[1], in_socket);
        sprintf(argv[2], out_socket);

        sleep(5);
        execl("/home/yagna_s/cs763/project/tkn_ring", "tkn_ring", argv[1], argv[2], buf
      }
    }

    /*********************************************/
```

```
/********* TKN_RING.C ******************
This is a simple version of token ring protocol written to exhibit the
use of the distributed system interface given by pilot.c and server.c
*****************************************/

#include <stdio.h>
#include <errno.h>
#include <sys/time.h>
#include <math.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <sys/uio.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netdb.h>

#define  m1  259200
#define  ia1 7141
#define  ic1  54773
#define  rm1 3.8580247e-6
#define  m2  134456
#define  ia2  8121
#define  ic2 28411
#define  rm2 7.4373773e-6
#define m3  243000
#define ia3  4561
#define  ic3  51349

     int idum;
     int glix1, glix2, glix3;
     float glr[98];
     float r;
float randl();

#define MAX_RANDOM_NUMBER 2147483647
#define MAXHOSTNAME 80

#define NOTE "How_do_you_do"
#define MAXLEN 1024

char names[20][40], myname[40], localhost[40];
int maxnodes = 0, insocket, outsocket;
char token[1024];
char source[40], dest[40], data[500];
int ack, is_tkn, msg_id, next_msg = 0;
char msg[MAXLEN];

FILE *fp, *fopen();

main(argc, argv)

int argc;
char **argv;

{
  int i;
  void decipher();
  struct  hostent *hp, *gethostbyname();
  char *tmp_buf, buf[80];

  strcpy(token, "1\nhorsa.cs.odu.edu\noffa.cs.odu.edu\nNOTE\n0\n1\n");
```

```
    fp = fopen("/tmp/log", "w+");

    insocket = atoi(argv[1]);
    outsocket = atoi(argv[2]);

    tmp_buf = argv[3];
    i = 0;
    while(sscanf(tmp_buf, "%s", buf) >0){
      strcpy(names[i++], buf);
      tmp_buf += strlen(buf) + 1;
      maxnodes++;
    }
  /*
    i = 3;
    while(i < argc){
      strcpy(names[i-3], argv[i]);
      i++;
      printf("%s\n", argv[i]);
    }
    maxnodes = argc - 3;
  */
    gethostname(localhost, MAXHOSTNAME);

    hp = gethostbyname(localhost);

    strcpy(myname, hp->h_name);

    for (;;){
      sleep(2);
      if(recv(insocket, msg, MAXLEN, 0) < 0){
        printf("%s\n", msg);
        perror("100:receiving on socket");
        close(insocket);
        exit(100);
      }

  /*      strcpy(msg, "0\nhorsa.cs.odu.edu\naelle.cs.odu.edu\nNOTE\n0\n1\n");
  */
      if(!strcmp(msg, "")){
        if(send(outsocket, msg, MAXLEN, 0) < 0)
          break;
      }
      decipher();
      if(is_tkn)
        handle_token();
      else
        process_msg();
    }
    fclose(fp);
    close(insocket);
    close(outsocket);
  }
  /*****************************************************************************/
  handle_token()
  {
    double k;
  /*
    k = (double)(getRandInt(1, 100)/100);
  */
    k = (double)random()/(double)MAX_RANDOM_NUMBER;

    fprintf(fp, "Received token\n");
    fflush(fp);

    if (k > 0.0)
      send_msg();
```

```
    else
      send_token();
  }

  /*****************************************************************************/

  process_msg()
  {
    if (!strcmp(dest, myname))          /* this is sent for me*/
      process_data();
    else if (!strcmp(source, myname))
      handle_ack();
    else{
      fprintf(fp, "%s: passing message: %d: of :%s : to: %s \n",
              myname, msg_id, source, dest);
      if(send(outsocket, msg, MAXLEN, 0) < 0){
        perror("50:sending Ack");
        exit(50);
      }
    }
  }

  /*****************************************************************************/
  /*
  int getRandInt(b)

  int b;

  {
   double seed = 987654321;

     return (((seed)*100) % b);
     }
  */
  /*****************************************************************************/

  send_msg()

  {

    char message[MAXLEN], *tmp_buf;
    int k;
    tmp_buf = message;

    sprintf(tmp_buf, "%d", 0);
    tmp_buf += sizeof(char); tmp_buf[0] = '\n'; tmp_buf++;
    sprintf(tmp_buf, "%s", myname);
    tmp_buf += strlen(myname); tmp_buf[0] = '\n';tmp_buf++;

    do
      k = getRandInt(0,maxnodes);
    while(!strcmp(names[k], myname));

    sprintf(tmp_buf, "%s", names[k]);
    tmp_buf += strlen(names[k]); tmp_buf[0] = '\n';tmp_buf++;
    sprintf(tmp_buf, "%s", NOTE);
    tmp_buf += strlen(NOTE); tmp_buf[0] = '\n';tmp_buf++;
    sprintf(tmp_buf, "%d", 0);
    tmp_buf += sizeof(char); tmp_buf[0] = '\n';tmp_buf++;
    sprintf(tmp_buf, "%d\n", ++next_msg);

    /*Now ready to send this message off*/

    if(send(outsocket, message, MAXLEN, 0) < 0){
      perror("200 nding message");
      exit(200);
```

```
    }
    fprintf(fp, "Sent message:%d: to :%s\n", (next_msg - 1), names[k]);
    fflush(fp);

    /*Anyway we have to pass token too*/

    if(send(outsocket, token, MAXLEN, 0) < 0){
      perror("300:sending token");
      exit(300);
    }
    fprintf(fp, "Sent Token too\n");
    fflush(fp);

  }


  /*****************************************************************************/


  send_token()
  {
    if(send(outsocket, token, MAXLEN, 0) < 0){
      perror("sending token on the outsocket");
      exit(300);
    }
    fprintf(fp, "Sent Token\n");
    fflush(fp);

  }

  /*****************************************************************************/


  process_data()
  {
    char *tmp_buf;

    fprintf(fp, ":%s: received message: %d: from :%s:\n",
                myname, msg_id, source);
    fflush(fp);

    tmp_buf = msg;
    tmp_buf += sizeof(char)+strlen(source)+strlen(dest)+strlen(data)+4;
    tmp_buf[0] = '1';

    fprintf(fp, "sending ack for msg:%d\n", msg_id);
    fflush(fp);

    if(send(outsocket, msg, MAXLEN, 0) < 0){
      perror("300:sending Ack");
      exit(300);
    }
  }

  /*****************************************************************************/


  handle_ack(){
    fprintf(fp, "%s: received ack from: %s: for message :%d\n",
            myname, dest, msg_id);
    fflush(fp);
  }

  /*****************************************************************************
                  random number generator between 0 and 1
```

```
                                        random()
*****************************************************************************/


float randl(idum)
     int *idum;
{

     int j;

     float ret;
     if(*idum < 0 )
       {
            glix1 = (ic1 - *idum) % m1;
            glix1 = (ia1 * glix1 + ic1) % m1;
            glix2 = glix1 % m2;
            glix1 = (ia1 * glix1 + ic1) % m1;
            glix3 = glix1 % m3;

            for(j = 1;j< 97;j++)
              {
                   glix1  = (ia1 * glix1 + ic1) % m1;
                   glix2  = (ia2 * glix2 + ic2) % m2;
                   glr[j] = (glix1 + glix2 * rm2) * rm1;
              }
            *idum = 1;
       }

     glix1 = (ia1 * glix1 + ic1) % m1;
     glix2 = (ia2 * glix2 + ic2) % m2;
     glix3 = (ia3 * glix3 + ic3) % m3;

     j = (int) (1 + (97 * glix3) / m3 );
     if( (j > 97) ||(j < 1) )
       {
            printf("halted in random\n");
       }
     ret = glr[j];
     glr[j] = (glix1 + glix2 * rm2) * rm1;
     return(ret);
}
/*****************************************************************************
 *                      returns random intege between a and b
 *                                      getRandInt()
 *****************************************************************************/

int getRandInt(a,b)
     int a;
     int b;
{
  int i;

  for(i=0;i<100;i++)
    randl(&idum);
  return (int)( a + (b-a+1)*( randl(&idum) ) );
}
/*****************************************************************************/
void decipher()
{
  char *tmp_buf;
  char temp[4];

  tmp_buf = msg;
  sscanf(tmp_buf, "%d", &is_tkn);
/*  is_tkn = atoi(temp);*/
  if(!is_tkn){
```

```
        tmp_buf += sizeof(char);
        tmp_buf++;
        sscanf(tmp_buf, "%s", source);
        tmp_buf += strlen(source);
        tmp_buf++;
        sscanf(tmp_buf, "%s", dest);
        tmp_buf += strlen(dest);
        tmp_buf++;
        sscanf(tmp_buf, "%s", data);
        tmp_buf += strlen(data);
        tmp_buf++;
        sscanf(tmp_buf, "%d", &ack);
    /*    ack = atoi(temp);*/
        tmp_buf += sizeof(char);
        tmp_buf++;
        sscanf(tmp_buf, "%d", &msg_id);
      }
    }
    /*************************************************************/
```